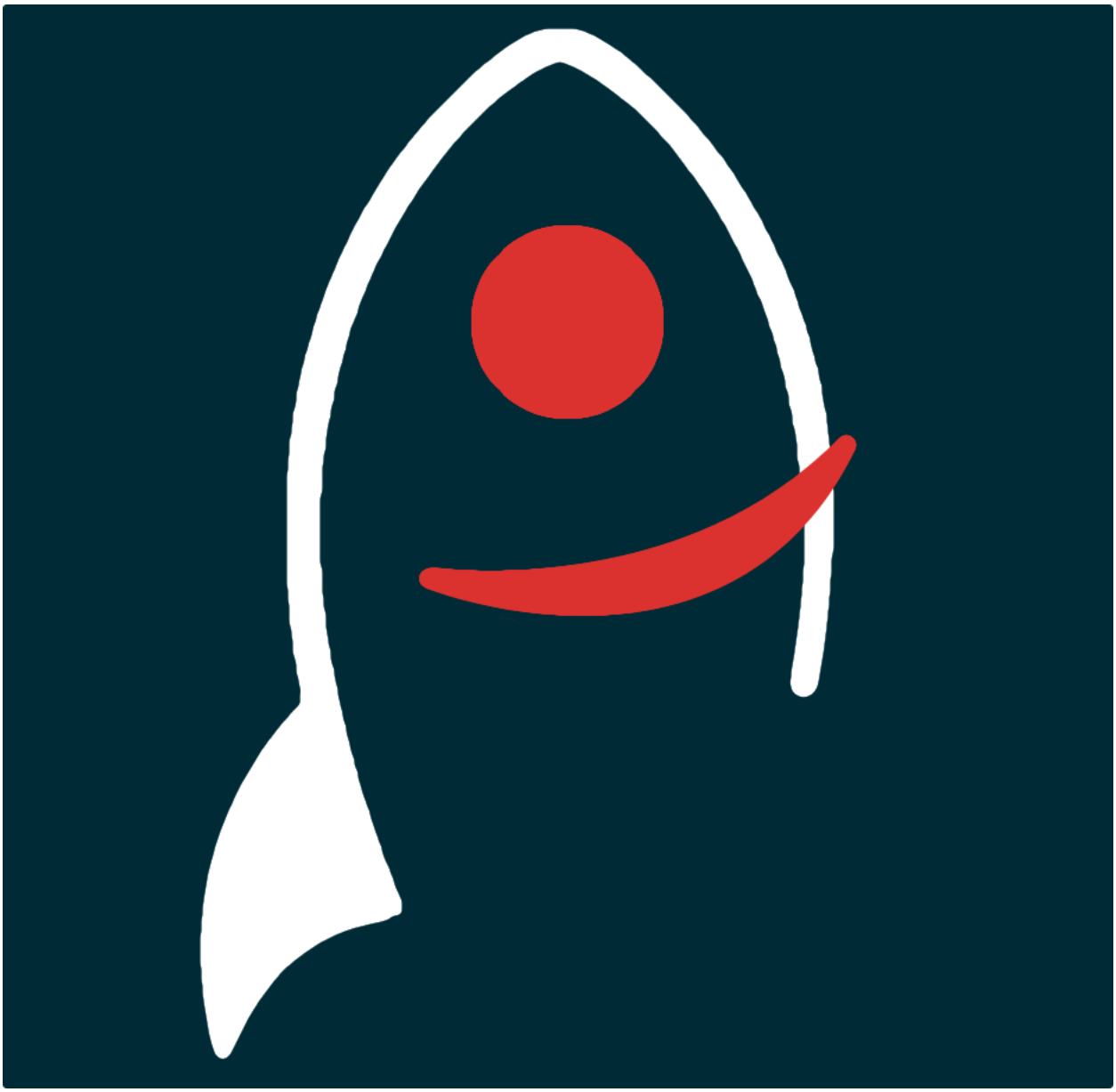


fundamentals Documentation

Release v2.4.1

Dave Young

2023

TABLE OF CONTENTS

1 Features **3**

2 How to cite fundamentals **5**
- 2.1 Installation . . . 5
  - 2.1.1 Development . . . 5
- 2.2 Initialisation . . . 6
  - 2.2.1 Modifying the Settings . . . 6
  - 2.2.2 Basic Python Setup . . . 6
- 2.3 Todo List . . . 6
- 2.4 Release Notes . . . 7

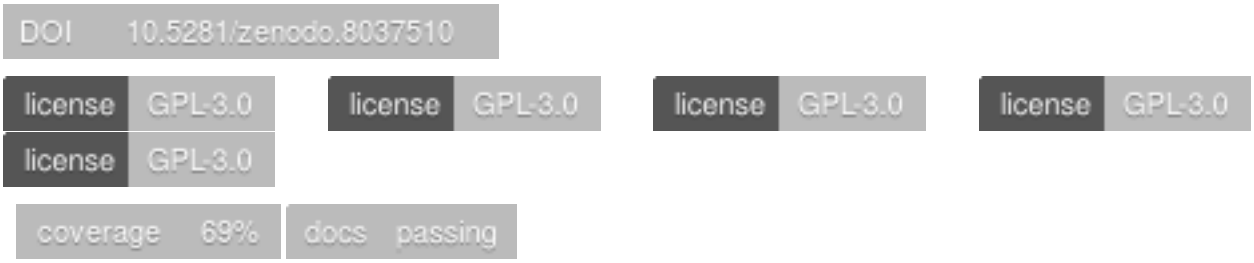
3 API Reference **9**
- 3.1 Modules . . . 9
  - 3.1.1 commonutils *(module)* . . . 9
  - 3.1.2 download *(module)* . . . 9
  - 3.1.3 files *(module)* . . . 10
  - 3.1.4 mysql *(module)* . . . 10
  - 3.1.5 nose2_plugins *(module)* . . . 11
  - 3.1.6 renderer *(module)* . . . 11
  - 3.1.7 stats *(module)* . . . 11
  - 3.1.8 logs *(module)* . . . 12
  - 3.1.9 times *(module)* . . . 12
- 3.2 Classes . . . 13
  - 3.2.1 daemonise *(class)* . . . 13
  - 3.2.2 fileChunker *(class)* . . . 15
  - 3.2.3 GroupWriteRotatingFileHandler *(class)* . . . 15
  - 3.2.4 emptyLogger *(class)* . . . 18
  - 3.2.5 database *(class)* . . . 18
  - 3.2.6 sqlite2mysql *(class)* . . . 19
  - 3.2.7 yaml_to_database *(class)* . . . 20
  - 3.2.8 list_of_dictionaries *(class)* . . . 22
  - 3.2.9 tools *(class)* . . . 27
  - 3.2.10 utKit *(class)* . . . 29
- 3.3 Functions . . . 30
  - 3.3.1 getpackagepath (*function*) . . . 31
  - 3.3.2 append_now_datestamp_to_filename (*function*) . . . 31
  - 3.3.3 extract_filename_from_url (*function*) . . . 31
  - 3.3.4 get_now_datetime_filestamp (*function*) . . . 32
  - 3.3.5 multiobject_download (*function*) . . . 32
  - 3.3.6 list_of_dictionaries_to_mysql_inserts (*function*) . . . 33

3.3.7 recursive_directory_listing (*function*) . . . 34
3.3.8 tag (*function*) . . . 34
3.3.9 fmultiprocess (*function*) . . . 35
3.3.10 console_logger (*function*) . . . 36
3.3.11 setup_dryx_logging (*function*) . . . 36
3.3.12 convert_dictionary_to_mysql_table (*function*) . . . 37
3.3.13 directory_script_runner (*function*) . . . 39
3.3.14 get_database_table_column_names (*function*) . . . 40
3.3.15 insert_list_of_dictionaries_into_database_tables (*function*) . . . 41
3.3.16 readquery (*function*) . . . 42
3.3.17 setup_database_connection (*function*) . . . 42
3.3.18 table_exists (*function*) . . . 43
3.3.19 writequery (*function*) . . . 43
3.3.20 rolling_window_sigma_clip (*function*) . . . 44
3.3.21 calculate_time_difference (*function*) . . . 45
3.3.22 datetime_relative_to_now (*function*) . . . 45
3.3.23 get_now_sql_datetime (*function*) . . . 45
3.4 A-Z Index . . . 46

4 Release Notes **49**

Python Module Index **51**

Index **53**



Some fundamental tools required by most self-respecting python-packages bundled in one place. Very opinionated project setup tools including logging, plain-text settings files and database connections.

Documentation for fundamentals is hosted by Read the Docs (development version and master version). The code lives on github. Please report any issues you find here.

CHAPTER

ONE

FEATURES

-

CHAPTER

TWO

HOW TO CITE FUNDAMENTALS

If you use `fundamentals` in your work, please cite using the following BibTeX entry:

```
@software{Young_fundamentals,
    author = {Young, David R.},
    doi = {10.5281/zenodo.8037510},
    license = {GPL-3.0-only},
    title = ,
    url = {https://github.com/thespacedoctor/fundamentals}
}
```

2.1 Installation

The easiest way to install fundamentals is to use `pip` (here we show the install inside of a conda environment):

```
conda create -n fundamentals python=3.7 pip
conda activate fundamentals
pip install fundamentals
```

Or you can clone the github repo and install from a local version of the code:

```
git clone git@github.com:thespacedoctor/fundamentals.git
cd fundamentals
python setup.py install
```

To upgrade to the latest version of fundamentals use the command:

```
pip install fundamentals --upgrade
```

To check installation was successful run `fundamentals -v`. This should return the version number of the install.

2.1.1 Development

If you want to tinker with the code, then install in development mode. This means you can modify the code from your cloned repo:

```
git clone git@github.com:thespacedoctor/fundamentals.git
cd fundamentals
python setup.py develop
```

Pull requests are welcomed!

2.2 Initialisation

Before using fundamentals you need to use the `init` command to generate a user settings file. Running the following creates a yaml settings file in your home folder under `~/.config/fundamentals/fundamentals.yaml`:

```
fundamentals init
```

The file is initially populated with fundamentals' default settings which can be adjusted to your preference.

If at any point the user settings file becomes corrupted or you just want to start afresh, simply trash the `fundamentals.yaml` file and rerun `fundamentals init`.

2.2.1 Modifying the Settings

Once created, open the settings file in any text editor and make any modifications needed.

2.2.2 Basic Python Setup

If you plan to use `fundamentals` in your own scripts you will first need to parse your settings file and set up logging etc. `fundamentals` itself can give you a logger, a settings dictionary and a database connection (if connection details given in settings file):

```
## SOME BASIC SETUP FOR LOGGING, SETTINGS ETC
from fundamentals import tools
from os.path import expanduser
home = expanduser("~")
settingsFile  = home + "/.config/fundamentals/fundamentals.yaml"
su = tools(
    arguments={"settingsFile": settingsFile},
    docString=__doc__,
)
arguments, settings, log, dbConn = su.setup()
```

2.3 Todo List

Todo:

- add a tutorial about `sqlite2mysql` to documentation

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/fundamentals/envs/master/lib/python3.7/site-packages/fundamentals-2.4.1-py3.7.egg/fundamentals/mysql/sqlite2mysql.py:docstring of fundamentals.mysql.sqlite2mysql.sqlite2mysql, line 19.)

Todo:

- nice!

(The *original entry* is located in /home/docs/checkouts/readthedocs.org/user_builds/fundamentals/checkouts/master/docs/source/_templa line 1.)

2.4 Release Notes

v2.4.1 - May 24, 2023

- **FIXED** small bug fixes to daemonise code

v2.4.0 - May 24, 2023

- **FEATURE** added a class to help daemonise code

v2.3.12 - May 10, 2022

- **FIXED** doc fixes

v2.3.11 - March 17, 2022

- **FIXED**: deadlocked connections now attempt to reconnect.

v2.3.10 - March 7, 2022

- **ENHANCEMENT:** can now turn off multiprocessing with the `turnOffMP` parameter of `fmultiprocessing`. Needed for full profiling of code.

v2.3.9 - November 8, 2021

- **FIXED:** moved depreciated calls to yaml `load` to `safe_load`

v2.3.8 - September 29, 2021

- **ENHANCEMENT:** fundamentals is now on conda-forge

v2.3.7 - September 27, 2021

- **ENHANCEMENT:** some speed improvements in multi-downloads

v2.3.6 - August 16, 2021

- **FIXED:** no longer crashing for scripts where no settings file is passed in via CL arguments
- **FIXED:** database credentials can now be passed to the command-line again

v2.3.5 - July 30, 2021

- **FEATURE:** code bases using fundamentals can now include a 'advanced_settings.yaml' file at the root of the project which will be read before the user settings file. User settings trump settings in this 'advanced_settings.yaml' file. The purpose is to be able to have hidden/development settings.

v2.3.4 - March 16, 2021

- **ENHANCEMENT:** added loop to reattempt timed-out queries (up to 60 times)

v2.3.2 - February 23, 2021

- **FIXED:** Logger was being set from default settings file even if a custom settings file given from command line

v2.3.1 - January 6, 2021

- **FIXED:** astropy import causing grief with other package installs. Move to with function instead of module level import.

v2.3.0 - January 1, 2021

- **FEATURE:** added a stats subpackage with a `rolling_window_sigma_clip` function

v2.2.9 - December 3, 2020

- **FIXED:** relative time reporting (python 2>3ism)

v2.2.8 - November 12, 2020

- **fixed:** logging levels

v2.2.7 - November 10, 2020

- **fixed:** mysql port connection issue (with MaxScale proxy)

v2.2.6 - November 9, 2020

- **fixed:** syntax error

v2.2.5 - November 2, 2020

- **enhancement:** adding colour to logs
- **enhancement:** addition of port in database connection settings
- **fixed:** replacing depreciated 'is" syntax with ==

v2.2.4 - May 28, 2020

- **enhancement:** allowing '~' as home directory for filepaths in all settings file parameters - will be converted when initially read

v2.2.3 - May 26, 2020

- **fixed:** delimiters catered for in sql scripts

v2.2.2 - May 25, 2020

- **refactor:**`list_of_dictionaries` now returns bytes decoded into UTF-8 string when rendered to other mark-up flavour.
- **refactor:** moved module level numpy imports so that packages with fundamentals as a dependency do not have numpy as a needless dependency

v2.2.1 - May 13, 2020

- **fixed:**`directory_script_runner` function `databaseName` parameter changed to be optional

v2.2.0 - May 13, 2020

- **feature:** new `execute_mysql_script` function that allows execution of a sql script directly from file
- **refactor:** added a dbConn to the directory script runner

v2.1.7 - May 4, 2020

- **fixed:** inspect module depreciation warnings stopped
- **fixed:** MySQL error messages printed to stdout (previously hidden)

v2.1.3 - April 17, 2020

- **enhancement:** cleaned up docs theme and structure. More documentation to come.

CHAPTER

THREE

API REFERENCE

3.1 Modules

`fundamentals.commonutils`	*Common tools used thoughout the fundamentals package*
`fundamentals.download`	*Tools used to perform download of files and HTML pages from the web*
`fundamentals.files`	*Tools for working with files and folders*
`fundamentals.mysql`	*Some handy mysql database query and insertion tools*
`fundamentals.nose2_plugins`	*plugins for nose2 unit testing*
`fundamentals.renderer`	*Render python objects as various list and markup formats *
`fundamentals.stats`	*Some reusable statistic functions*
`fundamentals.logs`	*Logger setup for python projects*
`fundamentals.times`	*Some time functions to be used with logging etc*

3.1.1 commonutils *(module)*

Common tools used thoughout the fundamentals package

Functions

`getpackagepath`()	*Get the root path for this python package*

3.1.2 download *(module)*

Tools used to perform download of files and HTML pages from the web

Functions

append_now_datestamp_to_filename(log, filename)	*append the current datestamp to the end of the filename (before the extension).*
extract_filename_from_url(log, url)	*get the filename from a URL.*
get_now_datetime_filestamp([longTime])	*A datetime stamp to be appended to the end of filenames: 'YYYYMMDDtHHMMSS'*
multiobject_download(urlList, ... [, ...])	*get multiple url documents and place them in specified download directory/directories*

3.1.3 files *(module)*

Tools for working with files and folders

Classes

fileChunker(filepath, batchSize)	*The fileChunker iterator - iterate over large line-based files to reduce memory footprint*

Functions

list_of_dictionaries_to_mysql_inserts(log, ...)	Convert a python list of dictionaries to pretty csv output
recursive_directory_listing(log, baseFolderPath)	*list directory contents recursively.*
tag(log, filepath[, tags, rating, wherefrom])	Add tags and ratings to your macOS files and folders

3.1.4 mysql *(module)*

Some handy mysql database query and insertion tools

Classes

database(log[, dbSettings, autocommit])	*a database object that can setup up a ssh tunnel (optional) and a database connection*
sqlite2mysql(log, pathToSqlite[, ...])	*Take a sqlite database file and copy the tables within it to a MySQL database*
yaml_to_database(log, dbConn[, ...])	*Take key-values from yaml files including a table-name(s) and add them to a mysql database table*

Functions

convert_dictionary_to_mysql_table(log, ...)	convert dictionary to mysql table
directory_script_runner(log, ...[, dbConn, ...])	A function to run all mysql scripts in a given directory (in a modified date order, oldest first) and then act on the script files in accordance with the succcess or failure of their execution
execute_mysql_script(pathToScript, dbConn, log)	*execute a mysql script given its file path and return the success or failure status of the execution*
get_database_table_column_names(dbConn, log, ...)	get database table column names
insert_list_of_dictionaries_into_database_tables(...)	insert list of dictionaries into database tables
readquery(sqlQuery, dbConn, log[, quiet])	Given a mysql query, read the data from the database and return the results as a list of dictionaries (database rows)
setup_database_connection(pathToYamlFile)	*Start a database connection using settings in yaml file*
table_exists(dbConn, log, dbTableName)	*Probe a database to determine if a given table exists*
writequery(log, sqlQuery, dbConn[, Force, ...])	*Execute a MySQL write command given a sql query*

3.1.5 nose2_plugins *(module)*

plugins for nose2 unit testing

3.1.6 renderer *(module)*

*Render python objects as various list and markup formats *

Classes

list_of_dictionaries(log, listOfDictionaries)	*The dataset object is a list of python dictionaries.

3.1.7 stats *(module)*

Some reusable statistic functions

Functions

rolling_window_sigma_clip(log, array, ...)	*given a sorted list of values, median sigma-clip values based on a window of values either side of each value (rolling window) and return the array mask*

3.1.8 logs *(module)*

Logger setup for python projects

Author David Young

Classes

GroupWriteRotatingFileHandler(filename[, ...])	*rotating file handler for logging*
emptyLogger()	*A fake logger object so user can set ``log=False`` if required*
object()	The most base type

Functions

console_logger([level])	*Setup and return a console logger*
setup_dryx_logging(yaml_file)	*setup dryx style python logging*

3.1.9 times *(module)*

Some time functions to be used with logging etc

Author David Young

Classes

str([object])	str(bytes_or_buffer[, encoding[, errors]]) -> str

Functions

calculate_time_difference(startDate, endDate)	*Return the time difference between two dates as a string*
datetime_relative_to_now(date)	*convert date to a relative datetime (e.g.
get_now_sql_datetime()	*A datetime stamp in MySQL format: 'YYYY-MM-DDTHH:MM:SS'*

3.2 Classes

fundamentals.daemonise	A class to daemonise a python code
fundamentals.files.fileChunker	*The fileChunker iterator - iterate over large line-based files to reduce memory footprint*
fundamentals.logs.GroupWriteRotatingFileHandler	*rotating file handler for logging*
fundamentals.logs.emptyLogger	*A fake logger object so user can set ``log=False`` if required*
fundamentals.mysql.database	*a database object that can setup up a ssh tunnel (optional) and a database connection*
fundamentals.mysql.sqlite2mysql	*Take a sqlite database file and copy the tables within it to a MySQL database*
fundamentals.mysql.yaml_to_database	*Take key-values from yaml files including a table-name(s) and add them to a mysql database table*
fundamentals.renderer.list_of_dictionaries	*The dataset object is a list of python dictionaries.*
fundamentals.tools	*common setup methods & attributes of the main function in cl-util*
fundamentals.utKit	*Default setup for fundamentals style unit-testing workflow (all tests base on nose module)*

3.2.1 daemonise *(class)*

class daemonise(*log*, *name*, ***akws*)

Bases: `object`

A class to daemonise a python code

Key Arguments:

- `log` – logger
- `name` – a name for the daemon (e.g. python package name)

Usage:

Add something like this to your command-line usage:

```
Usage:
myCommand (start|stop|restart|status)

Options:
    start                start the myCommand daemon
    stop                 stop the myCommand daemon
    restart              restart the myCommand daemon
    status               print the staus of the myCommand daemon
```

and then when executing the commands:

```
from fundamental import daemonise
class myDaemon(daemonise):
    def action(
            self,
            **kwargs):
```

(continues on next page)

(continued from previous page)

```
        self.log.info('starting the ``action`` method')

        anotherParameter = kwargs["anotherParameter"]

        import time
        while True:
            print(f"OVERRIDE ACTION - {anotherParameter}")
            time.sleep(3)

        self.log.info('completed the ``action`` method')
        return None

d = myDaemon(log=log, name="gocart", anotherParameter=42.)

if a['start']:
    d.start()
elif a['stop']:
    d.stop()
elif a['restart']:
    d.restart()
elif a['status']:
    d.status()
```

Replace `**akws` with any keywords you need.

Methods

`action`(**akws)	*the code to execute in daemon mode, this method should be overriden to execute novel code*
`cleanup`(signum, frame)	*the code to run when daemon is killed*
`restart`()	*stop and start the daemon*
`start`()	start the daemonise running
`status`()	*print the status of the daemon*
`stop`()	*stop the daemon and cleanup*

action(***akws*)
the code to execute in daemon mode, this method should be overriden to execute novel code

cleanup(*signum*, *frame*)
the code to run when daemon is killed

restart()
stop and start the daemon

start()
start the daemonise running

status()
print the status of the daemon

stop()
stop the daemon and cleanup

3.2.2 fileChunker *(class)*

class fileChunker(*filepath*, *batchSize*)

Bases: `object`

The fileChunker iterator - iterate over large line-based files to reduce memory footprint

Key Arguments

- `filepath` – path to the large file to iterate over
- `batchSize` – size of the chunks to return in lines

Usage

To setup your logger, settings and database connections, please use the `fundamentals` package (see tutorial here).

To initiate a fileChunker iterator and then process the file in batches of 100000 lines, use the following:

```
from fundamentals.files import fileChunker
fc = fileChunker(
    filepath="/path/to/large/file.csv",
    batchSize=100000
)
for i in fc:
    print len(i)
```

Methods

—

3.2.3 GroupWriteRotatingFileHandler *(class)*

class GroupWriteRotatingFileHandler(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*)

Bases: `logging.handlers.RotatingFileHandler`

rotating file handler for logging

Methods

acquire()	Acquire the I/O thread lock.
addFilter(filter)	Add the specified filter to this handler.
close()	Closes the stream.
createLock()	Acquire a thread lock for serializing access to the underlying I/O.
doRollover()	Do a rollover, as described in **init**().
emit(record)	Emit a record.
filter(record)	Determine if a record is loggable by consulting all the filters.
flush()	Flushes the stream.
format(record)	Format the specified record.

continues on next page

Table 17 – continued from previous page

get_name()	
handle(record)	Conditionally emit the specified logging record.
handleError(record)	Handle errors which occur during an emit() call.
release()	Release the I/O thread lock.
removeFilter(filter)	Remove the specified filter from this handler.
rotate(source, dest)	When rotating, rotate the current log.
rotation_filename(default_name)	Modify the filename of a log file when rotating.
setFormatter(fmt)	Set the formatter for this handler.
setLevel(level)	Set the logging level of this handler.
setStream(stream)	Sets the StreamHandler's stream to the specified value, if it is different.
set_name(name)	
shouldRollover(record)	Determine if rollover should occur.

Properties

name
terminator

acquire()
Acquire the I/O thread lock.

addFilter(*filter*)
Add the specified filter to this handler.

close()
Closes the stream.

createLock()
Acquire a thread lock for serializing access to the underlying I/O.

doRollover()
Do a rollover, as described in **init**().

emit(*record*)
Emit a record.

Output the record to the file, catering for rollover as described in doRollover().

filter(*record*)
Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

flush()
Flushes the stream.

format(*record*)
Format the specified record.

If a formatter is set, use it. Otherwise, use the default formatter for the module.

handle(*record*)
Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

handleError(*record*)
Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

release()
Release the I/O thread lock.

removeFilter(*filter*)
Remove the specified filter from this handler.

rotate(*source*, *dest*)
When rotating, rotate the current log.

The default implementation calls the 'rotator' attribute of the handler, if it's callable, passing the source and dest arguments to it. If the attribute isn't callable (the default is None), the source is simply renamed to the destination.

Parameters

- **source** – The source filename. This is normally the base filename, e.g. 'test.log'
- **dest** – The destination filename. This is normally what the source is rotated to, e.g. 'test.log.1'.

rotation_filename(*default_name*)
Modify the filename of a log file when rotating.

This is provided so that a custom filename can be provided.

The default implementation calls the 'namer' attribute of the handler, if it's callable, passing the default name to it. If the attribute isn't callable (the default is None), the name is returned unchanged.

Parameters **default_name** – The default name for the log file.

setFormatter(*fmt*)
Set the formatter for this handler.

setLevel(*level*)
Set the logging level of this handler. level must be an int or a str.

setStream(*stream*)
Sets the StreamHandler's stream to the specified value, if it is different.

Returns the old stream, if the stream was changed, or None if it wasn't.

shouldRollover(*record*)
Determine if rollover should occur.

Basically, see if the supplied record would cause the file to exceed the size limit we have.

3.2.4 emptyLogger *(class)*

class emptyLogger
Bases: object

A fake logger object so user can set ``log=False`` if required

Usage

```
if log == False:
    from fundamentals.logs import emptyLogger
    log = emptyLogger()
```

Methods

critical(argu)
debug(argu)
error(argu)
info(argu)
warning(argu)

3.2.5 database *(class)*

class database(*log*, *dbSettings=False*, *autocommit=True*)
Bases: object

a database object that can setup up a ssh tunnel (optional) and a database connection

Key Arguments

- log – logger
- dbSettings – a dictionary of database settings

Return

- dbConns – a database connection

Usage

Given a python dictionary that looks like this:

```
dbSettings = {
    'host': '127.0.0.1',
    'loginPath': 'atlasMovers',
    'user': 'monster',
    'tunnel': {
        'remote ip': 'psweb.mp.qub.ac.uk',
        'remote datbase host': 'dormammu',
        'remote user': 'monster',
        'port': 9006
```

(continues on next page)

(continued from previous page)

```
    },
    'password': 'myPass',
    'db': 'atlas_moving_objects'
}
```

`loginPath` and `tunnel` are optional, to setup the a database connection, run the following:

```
# SETUP ALL DATABASE CONNECTIONS
from fundamentals.mysql import database
dbConn = database(
    log=log,
    dbSettings=dbSettings
).connect()
```

Methods

`connect()`	*Connect to the database*

connect()
Connect to the database

Return

- `dbConn` – the database connection

See the class docstring for usage

3.2.6 sqlite2mysql *(class)*

class sqlite2mysql(*log*, *pathToSqlite*, *tablePrefix=''*, *settings=False*, *dbConn=False*)
Bases: `object`

Take a sqlite database file and copy the tables within it to a MySQL database

Key Arguments

- `log` – logger
- `settings` – the settings dictionary
- `pathToSqlite` – path to the sqlite database to transfer into the MySQL database
- `tablePrefix` – a prefix to add to all the tablename when converting to mysql. Default *""*
- `dbConn` – mysql database connection

Usage

To setup your logger, settings and database connections, please use the `fundamentals` package (see tutorial here).

To convert and import the content of a sqlite database into MySQL run the following:

Todo:

- add a tutorial about `sqlite2mysql` to documentation

```
from fundamentals.mysql import sqlite2mysql
converter = sqlite2mysql(
    log=log,
    settings=settings,
    pathToSqlite="/path/to/sqlite.db",
    tablePrefix="external"
)
converter.convert_sqlite_to_mysql()
```

Methods

`convert_sqlite_to_mysql()`	*copy the contents of the sqlite database into the mysql database*

convert_sqlite_to_mysql()
copy the contents of the sqlite database into the mysql database

See class docstring for usage

3.2.7 yaml_to_database *(class)*

class yaml_to_database(*log*, *dbConn*, *pathToInputDir=False*, *settings=False*, *deleteFiles=False*)
Bases: `object`

Take key-values from yaml files including a tablename(s) and add them to a mysql database table

Key Arguments

- `log` – logger
- `settings` – the settings dictionary
- `pathToInputDir` – path to the directory containing the yaml files that will be added to the database table(s). Default *False*
- `dbConn` – connection to database to add the content to
- `deleteFiles` - - delete the yamls files once their content has been added to the database. Default * False*

Usage

To setup your logger, settings and database connections, please use the `fundamentals` package (see tutorial here).

To initiate a `yaml2db` object, use the following:

```
from fundamentals.mysql import yaml_to_database
yaml2db = yaml_to_database(
    log=log,
    settings=settings,
    dbConn=dbConn,
    pathToInputDir="/path/to/yaml/directory",
    deleteFiles=False
)
```

And here's an example of the content in a yaml file that this `yaml2db` object can parse:

```
title: Why you should do most of your text editing in : Sublime Text | Sublime␣
↪Text Tips
url: http://sublimetexttips.com/why-you-should-do-most-of-your-text-editing-in-
↪sublime-text/?utm_source=drip&utm_medium=email&utm_campaign=editor-proliferation
kind: webpage
subtype: article
table: web_articles,podcasts
```

Methods

add_yaml_file_content_to_database(filepath)	*given a file to a yaml file, add yaml file content to database*
ingest()	*ingest the contents of the directory of yaml files into a database*

add_yaml_file_content_to_database (*filepath*, *deleteFile=False*)
given a file to a yaml file, add yaml file content to database

Key Arguments

- filepath – the path to the yaml file
- deleteFile – delete the yaml file when its content has been added to the database. Default *False*

Return

- None

Usage

To parse and import the contents of a single yaml file into the database, use the following:

```
from fundamentals.mysql import yaml_to_database
# PARSE YAML FILE CONTENTS AND ADD TO DATABASE
yaml2db = yaml_to_database(
    log=log,
    settings=settings,
    dbConn=dbConn
)
yaml2db.add_yaml_file_content_to_database(
    filepath=${1:"/path/to/file.yaml"},
    deleteFile=True
)
```

ingest ()
ingest the contents of the directory of yaml files into a database

Return

- None

Usage

To import an entire directory of yaml files into a database, use the following:

```
from fundamentals.mysql import yaml_to_database
yaml2db = yaml_to_database(
    log=log,
```

(continues on next page)

(continued from previous page)

```
    settings=settings,
    dbConn=dbConn,
    pathToInputDir="/path/to/yaml/directory",
    deleteFiles=False
)
yaml2db.ingest()
```

3.2.8 list_of_dictionaries *(class)*

class list_of_dictionaries(*log*, *listOfDictionaries*, *reDatetime=False*)

Bases: object

The dataset object is a list of python dictionaries. Using this class, the data can be rendered as various list and markup formats

Key Arguments

- log – logger
- listOfDictionaries – the list of dictionaries to render
- reDatetime – a pre-compiled datetime regex. Default *False*fss

Usage

To initialise the dataset object:

```
dataList = [
    {
        "owner": "daisy",
        "pet": "dog",
        "address": "belfast, uk"
    },
    {
        "owner": "john",
        "pet": "snake",
        "address": "the moon"
    },
    {
        "owner": "susan",
        "pet": "crocodile",
        "address": "larne"
    }

]

from fundamentals.renderer import list_of_dictionaries
dataSet = list_of_dictionaries(
    log=log,
    listOfDictionaries=dataList
)
```

Methods

csv([filepath])	*Render the data in CSV format*
json([filepath])	*Render the data in json format*
markdown([filepath])	*Render the data as a markdown table*
mysql(tableName[, filepath, createStatement])	*Render the dataset as a series of mysql insert statements*
reST([filepath])	*Render the data as a resturcturedText table*
table([filepath])	*Render the data as a plain text table*
yaml([filepath])	*Render the data in yaml format*

Properties

list	*Returns the original list of dictionaries*

csv(*filepath=None*)
Render the data in CSV format

Key Arguments

- filepath – path to the file to write the csv content to. Default *None*

Return

- renderedData – the data rendered in csv format

Usage

To render the data set as csv:

```
print(dataSet.csv())
```

```
owner,pet,address
daisy,dog,"belfast, uk"
john,snake,the moon
susan,crocodile,larne
```

and to save the csv rendering to file:

```
dataSet.csv("/path/to/myfile.csv")
```

json(*filepath=None*)
Render the data in json format

Key Arguments

- filepath – path to the file to write the json content to. Default *None*

Return

- renderedData – the data rendered as json

Usage

To render the data set as json:

```
print(dataSet.json())
```

```
[
    {
        "address": "belfast, uk",
        "owner": "daisy",
        "pet": "dog"
    },
    {
        "address": "the moon",
        "owner": "john",
        "pet": "snake"
    },
    {
        "address": "larne",
        "owner": "susan",
        "pet": "crocodile"
    }
]
```

and to save the json rendering to file:

```
dataSet.json("/path/to/myfile.json")
```

markdown(*filepath=None*)

Render the data as a markdown table

Key Arguments

- `filepath` – path to the file to write the markdown to. Default *None*

Return

- `renderedData` – the data rendered as a markdown table

Usage

To render the data set as a markdown table:

```
print(dataSet.markdown())
```

```
| owner   | pet         | address      |
|:-------|:-----------|:-------------|
| daisy   | dog         | belfast, uk  |
| john    | snake       | the moon     |
| susan   | crocodile   | larne        |
```

and to save the markdown table rendering to file:

```
dataSet.table("/path/to/myfile.md")
```

mysql(*tableName*, *filepath=None*, *createStatement=None*)

Render the dataset as a series of mysql insert statements

Key Arguments

- `tableName` – the name of the mysql db table to assign the insert statements to.
- `filepath` – path to the file to write the mysql inserts content to. Default *None* createStatement

Return

- `renderedData` – the data rendered mysql insert statements (string format)

Usage

```
print(dataSet.mysql("testing_table"))
```

this output the following:

```
INSERT INTO `testing_table` (address,dateCreated,owner,pet) VALUES ("belfast,␣
↪uk" ,"2016-09-14T16:21:36" ,"daisy" ,"dog")  ON DUPLICATE KEY UPDATE ␣
↪address="belfast, uk", dateCreated="2016-09-14T16:21:36", owner="daisy",␣
↪pet="dog" ;
INSERT INTO `testing_table` (address,dateCreated,owner,pet) VALUES ("the moon
↪" ,"2016-09-14T16:21:36" ,"john" ,"snake")  ON DUPLICATE KEY UPDATE ␣
↪address="the moon", dateCreated="2016-09-14T16:21:36", owner="john", pet=
↪"snake" ;
INSERT INTO `testing_table` (address,dateCreated,owner,pet) VALUES ("larne" ,
↪"2016-09-14T16:21:36" ,"susan" ,"crocodile")  ON DUPLICATE KEY UPDATE ␣
↪address="larne", dateCreated="2016-09-14T16:21:36", owner="susan", pet=
↪"crocodile" ;
```

To save this rendering to file use:

```
dataSet.mysql("testing_table", "/path/to/myfile.sql")
```

reST (*filepath=None*)

Render the data as a resturcturedText table

Key Arguments

- `filepath` – path to the file to write the table to. Default *None*

Return

- `renderedData` – the data rendered as a resturcturedText table

Usage

To render the data set as a resturcturedText table:

```
print(dataSet.reST())
```

```
+--------+------------+--------------+
| owner  | pet        | address      |
+========+============+==============+
| daisy  | dog        | belfast, uk  |
+--------+------------+--------------+
| john   | snake      | the moon     |
+--------+------------+--------------+
| susan  | crocodile  | larne        |
+--------+------------+--------------+
```

and to save the table rendering to file:

```
dataSet.reST("/path/to/myfile.rst")
```

table (*filepath=None*)

Render the data as a plain text table

Key Arguments

- `filepath` – path to the file to write the table to. Default *None*

Return

- `renderedData` – the data rendered as a plain text table

Usage

To render the data set as a plain text table:

```
print(dataSet.table())
```

```
+--------+------------+--------------+
| owner  | pet        | address      |
+========+============+==============+
| daisy  | dog        | belfast, uk  |
| john   | snake      | the moon     |
| susan  | crocodile  | larne        |
+--------+------------+--------------+
```

and to save the table rendering to file:

```
dataSet.table("/path/to/myfile.ascii")
```

yaml (*filepath=None*)
Render the data in yaml format

Key Arguments

- `filepath` – path to the file to write the yaml content to. Default *None*

Return

- `renderedData` – the data rendered as yaml

Usage

To render the data set as yaml:

```
print(dataSet.yaml())
```

```
- address: belfast, uk
  owner: daisy
  pet: dog
- address: the moon
  owner: john
  pet: snake
- address: larne
  owner: susan
  pet: crocodile
```

and to save the yaml rendering to file:

```
dataSet.json("/path/to/myfile.yaml")
```

property list
Returns the original list of dictionaries

Usage

dataSet.list

3.2.9 tools *(class)*

class tools(*arguments*, *docString*, *logLevel='WARNING'*, *options_first=False*, *projectName=False*, *distributionName=False*, *orderedSettings=False*, *defaultSettingsFile=False*, *quitIfRunning=True*)

Bases: `object`

common setup methods & attributes of the main function in cl-util

Key Arguments

- `dbConn` – mysql database connection
- `arguments` – the arguments read in from the command-line
- `docString` – pass the docstring from the host module so that docopt can work on the usage text to generate the required arguments
- `logLevel` – the level of the logger required. Default *DEBUG*. [DEBUG|INFO|WARNING|ERROR|CRITICAL]
- `options_first` – options come before commands in CL usage. Default *False*.
- `projectName` – the name of the project, used to create a default settings file in `~/.config/projectName/projectName.yaml`. Default *False*.
- `distributionName` – the distribution name if different from the projectName (i.e. if the package is called by anohter name on PyPi). Default *False*
- `tunnel` – will setup a ssh tunnel (if the settings are found in the settings file). Default *False*.
- `defaultSettingsFile` – if no settings file is passed via the doc-string use the default settings file in `~/.config/projectName/projectName.yaml` (don't have to clutter command-line with settings)

Usage

Add this to the `__main__` function of your command-line module

```
# setup the command-line util settings
from fundamentals import tools
su = tools(
    arguments=arguments,
    docString=__doc__,
    logLevel="DEBUG",
    options_first=False,
    projectName="myprojectName"
)
arguments, settings, log, dbConn = su.setup()
```

Here is a template settings file content you could use:

```
version: 1
database settings:
    db: unit_tests
    host: localhost
    user: utuser
    password: utpass
    tunnel: true

# SSH TUNNEL - if a tunnel is required to connect to the database(s) then add␣
↪setup here
```

(continues on next page)

(continued from previous page)

```
# Note only one tunnel is setup - may need to change this to 2 tunnels in the␣
↪future if
# code, static catalogue database and transient database are all on seperate␣
↪machines.
ssh tunnel:
    remote user: username
    remote ip: mydomain.co.uk
    remote datbase host: mydatabaseName
    port: 9002

logging settings:
    formatters:
        file_style:
            format: '* %(asctime)s - %(name)s - %(levelname)s (%(pathname)s >
↪%(funcName)s > %(lineno)d) - %(message)s  '
            datefmt: '%Y/%m/%d %H:%M:%S'
        console_style:
            format: '* %(asctime)s - %(levelname)s: %(pathname)s:%(funcName)s:
↪%(lineno)d > %(message)s'
            datefmt: '%H:%M:%S'
        html_style:
            format: '<div id="row" class="%(levelname)s"><span class="date">
↪%(asctime)s</span>   <span class="label">file:</span><span class="filename">
↪%(filename)s</span>   <span class="label">method:</span><span class="funcName">
↪%(funcName)s</span>   <span class="label">line#:</span><span class="lineno">
↪%(lineno)d</span> <span class="pathname">%(pathname)s</span>  <div class="right
↪"><span class="message">%(message)s</span><span class="levelname">%(levelname)s
↪</span></div></div>'
            datefmt: '%Y-%m-%d <span class= "time">%H:%M <span class= "seconds">
↪%Ss</span></span>'
    handlers:
        console:
            class: logging.StreamHandler
            level: DEBUG
            formatter: console_style
            stream: ext://sys.stdout
        file:
            class: logging.handlers.GroupWriteRotatingFileHandler
            level: WARNING
            formatter: file_style


                filename: /Users/Dave/.config/myprojectName/myprojectName.log
                mode: w+
                maxBytes: 102400
                backupCount: 1
        root:
            level: WARNING
            handlers: [file,console]
```

Methods

setup()	**Summary:**

setup()

Summary: *setup the attributes and return*

3.2.10 utKit *(class)*

class utKit(*moduleDirectory*)

Bases: object

Default setup for fundamentals style unit-testing workflow (all tests base on nose module)

Key Arguments

- moduleDirectory – the directory to the unit-testing test file

Usage

To use this kit within any of your unit-test modules add the following code before your test methods:

```
from fundamentals.utKit import utKit
# SETUP AND TEARDOWN FIXTURE FUNCTIONS FOR THE ENTIRE MODULE
moduleDirectory = os.path.dirname(__file__)
utKit = utKit(moduleDirectory)
log, dbConn, pathToInputDir, pathToOutputDir = utKit.setupModule()
utKit.tearDownModule()
```

Methods

get_project_root()	*Get the root of the ``python`` package - useful for getting files in the root directory of a project*
refresh_database()	*Refresh the unit test database*
setupModule()	*The setupModule method*
tearDownModule()	*The tearDownModule method*

get_project_root()

Get the root of the ``python`` package - useful for getting files in the root directory of a project

Return

- rootPath – the root path of a project

refresh_database()

Refresh the unit test database

setupModule()

The setupModule method

Return

- log – a logger
- dbConn – a database connection to a test database (details from yaml settings file)

- `pathToInputDir` – path to modules own test input directory
- `pathToOutputDir` – path to modules own test output directory

tearDownModule()
The tearDownModule method

3.3 Functions

fundamentals.commonutils.getpackagepath	*Get the root path for this python package*
fundamentals.download.append_now_datestamp_to_filename	*append the current datestamp to the end of the filename (before the extension).*
fundamentals.download.extract_filename_from_url	*get the filename from a URL.*
fundamentals.download.get_now_datetime_filestamp	*A datetime stamp to be appended to the end of file-names: 'YYYYMMDDtHHMMSS'*
fundamentals.download.multiobject_download	*get multiple url documents and place them in specified download directory/directories*
fundamentals.files.list_of_dictionaries_to_mysql_inserts	Convert a python list of dictionaries to pretty csv output
fundamentals.files.recursive_directory_listing	*list directory contents recursively.*
fundamentals.files.tag	Add tags and ratings to your macOS files and folders
fundamentals.fmultiprocess	multiprocess pool
fundamentals.logs.console_logger	*Setup and return a console logger*
fundamentals.logs.setup_dryx_logging	*setup dryx style python logging*
fundamentals.mysql.convert_dictionary_to_mysql_table	convert dictionary to mysql table
fundamentals.mysql.directory_script_runner	A function to run all mysql scripts in a given directory (in a modified date order, oldest first) and then act on the script files in accordance with the succcess or failure of their execution
fundamentals.mysql.get_database_table_column_names	get database table column names
fundamentals.mysql.insert_list_of_dictionaries_into_database_tables	insert list of dictionaries into database tables
fundamentals.mysql.readquery	Given a mysql query, read the data from the database and return the results as a list of dictionaries (database rows)
fundamentals.mysql.setup_database_connection	*Start a database connection using settings in yaml file*
fundamentals.mysql.table_exists	*Probe a database to determine if a given table exists*
fundamentals.mysql.writequery	*Execute a MySQL write command given a sql query*
fundamentals.stats.rolling_window_sigma_clip	*given a sorted list of values, median sigma-clip values based on a window of values either side of each value (rolling window) and return the array mask*
fundamentals.times.calculate_time_difference	*Return the time difference between two dates as a string*
fundamentals.times.datetime_relative_to_now	*convert date to a relative datetime (e.g.

continues on next page

Table 27 – continued from previous page

fundamentals.times.get_now_sql_datetime	*A datetime stamp in MySQL format: 'YYYY-MM-DDTHH:MM:SS'*

3.3.1 getpackagepath (*function*)

getpackagepath()
Get the root path for this python package

Used in unit testing code

3.3.2 append_now_datestamp_to_filename (*function*)

append_now_datestamp_to_filename(*log*, *filename*, *longTime=False*)
append the current datestamp to the end of the filename (before the extension).

Key Arguments

- `log` – logger
- `filename` – the filename
- `longTime` – use a longer time-stmap. Default *False*

Return:

- `dsFilename` – datestamped filename

Usage

```
# APPEND TIMESTAMP TO THE FILENAME
from fundamentals.download import append_now_datestamp_to_filename
filename = append_now_datestamp_to_filename(
        log=log,
        filename="some_filename.html",
        longTime=True
)

# OUTPUT
# 'some_filename_20160316t154123749472.html'
```

3.3.3 extract_filename_from_url (*function*)

extract_filename_from_url(*log*, *url*)
get the filename from a URL.

Will return 'untitled.html', if no filename is found.

Key Arguments

- `url` – the url to extract filename from

Returns:

- `filename` – the filename

Usage

```
from fundamentals.download import extract_filename_from_url
name = extract_filename_from_url(
    log=log,
    url="https://en.wikipedia.org/wiki/Docstring"
)
print name
# OUT: Docstring.html
```

3.3.4 get_now_datetime_filestamp (*function*)

get_now_datetime_filestamp(*longTime=False*)

A datetime stamp to be appended to the end of filenames: 'YYYYMMDDtHHMMSS'

Key Arguments

- `longTime` – make time string longer (more change of filenames being unique)

Return

- `now` – current time and date in filename format

Usage

```
from fundamentals.download import get_now_datetime_filestamp
get_now_datetime_filestamp(longTime=False)
#Out: '20160316t154635'

get_now_datetime_filestamp(longTime=True)
#Out: '20160316t154644133638'
```

3.3.5 multiobject_download (*function*)

multiobject_download(*urlList*, *downloadDirectory*, *log*, *timeStamp=True*, *timeout=180*, *concurrentDownloads=10*, *resetFilename=False*, *credentials=False*, *longTime=False*, *indexFilenames=False*)

get multiple url documents and place them in specified download directory/directories

Key Arguments

- `urlList` – list of document urls
- `downloadDirectory` – directory(ies) to download the documents to - can be one directory path or a list of paths the same length as urlList
- `log` – the logger
- `timestamp` – append a timestamp the name of the URL (ensure unique filenames)
- `longTime` – use a longer timestamp when appending to the filename (greater uniqueness)
- `timeout` – the timeout limit for downloads (secs)
- `concurrentDownloads` – the number of concurrent downloads allowed at any one time
- `resetFilename` – a string to reset all filenames to
- `credentials` – basic http credentials { 'username' : "...", "password", "..." }
- `indexFilenames` – prepend filenames with index (where url appears in urllist)

Return

- list of timestamped documents (same order as the input urlList)

Usage

```
# download the pages linked from the main list page
from fundamentals.download import multiobject_download
localUrls = multiobject_download(
    urlList=["https://www.python.org/dev/peps/pep-0257/","https://en.wikipedia.
↪org/wiki/Docstring"],
    downloadDirectory="/tmp",
    log="log",
    timeStamp=True,
    timeout=180,
    concurrentDownloads=2,
    resetFilename=False,
    credentials=False,  # { 'username' : "...", "password", "..." }
    longTime=True
)

print localUrls
# OUT: ['/tmp/untitled_20160316t160650610780.html',
# '/tmp/Docstring_20160316t160650611136.html']
```

3.3.6 list_of_dictionaries_to_mysql_inserts (*function*)

list_of_dictionaries_to_mysql_inserts(*log*, *datalist*, *tableName*)

Convert a python list of dictionaries to pretty csv output

Key Arguments

- `log` – logger
- `datalist` – a list of dictionaries
- `tableName` – the name of the table to create the insert statements for

Return

- `output` – the mysql insert statements (as a string)

Usage

```
from fundamentals.files import list_of_dictionaries_to_mysql_inserts
mysqlInserts = list_of_dictionaries_to_mysql_inserts(
    log=log,
    datalist=dataList,
    tableName="my_new_table"
)
print mysqlInserts
```

this output the following:

```
INSERT INTO `testing_table` (a_newKey,and_another,dateCreated,uniqueKey2,
↪uniquekey1) VALUES ("cool" ,"super cool" ,"2016-09-14T13:17:26" ,"burgers" ,
↪"cheese")  ON DUPLICATE KEY UPDATE  a_newKey="cool", and_another="super cool",␣
↪dateCreated="2016-09-14T13:17:26", uniqueKey2="burgers", uniquekey1="cheese" ;
...
...
```

3.3.7 recursive_directory_listing (*function*)

recursive_directory_listing(*log*, *baseFolderPath*, *whatToList='all'*)
list directory contents recursively.

Options to list only files or only directories.

Key Arguments

- `log` – logger
- `baseFolderPath` – path to the base folder to list contained files and folders recursively
- `whatToList` – list files only, durectories only or all ["files" | "dirs" | "all"]

Return

- `matchedPathList` – the matched paths

Usage

```
from fundamentals.files import recursive_directory_listing
theseFiles = recursive_directory_listing(
    log,
    baseFolderPath="/tmp"
)

# OR JUST FILE

from fundamentals.files import recursive_directory_listing
theseFiles = recursive_directory_listing(
    log,
    baseFolderPath="/tmp",
    whatToList="files"
)

# OR JUST FOLDERS

from fundamentals.files import recursive_directory_listing
theseFiles = recursive_directory_listing(
    log,
    baseFolderPath="/tmp",
    whatToList="dirs"
)
print theseFiles
```

3.3.8 tag (*function*)

tag(*log*, *filepath*, *tags=False*, *rating=False*, *wherefrom=False*)
Add tags and ratings to your macOS files and folders

Key Arguments

- `log` – logger
- `filepath` – the path to the file needing tagged
- `tags` – comma or space-separated string, or list of tags. Use `False` to leave file tags as they are. Use "" or [] to remove tags. Default *False*.

- `rating` – a rating to add to the file. Use 0 to remove rating or `False` to leave file rating as it is. Default *False*.
- `wherefrom` – add a URL to indicate where the file come from. Use `False` to leave file location as it is. Use "" to remove location. Default *False*.

Return

- None

Usage

To add any combination of tags, rating and a source URL to a file on macOS, use the following:

```
from fundamentals.files.tag import tag
tag(
    log=log,
    filepath="/path/to/my.file",
    tags="test,tags, fundamentals",
    rating=3,
    wherefrom="http://www.thespacedoctor.co.uk"
)
```

3.3.9 fmultiprocess (*function*)

fmultiprocess(*log*, *function*, *inputArray*, *poolSize=False*, *timeout=3600*, *turnOffMP=False*, ***kwargs*)

multiprocess pool

Key Arguments

- `log` – logger
- `function` – the function to multiprocess
- `inputArray` – the array to be iterated over
- `poolSize` – limit the number of CPU that are used in multiprocess job
- `timeout` – time in sec after which to raise a timeout error if the processes have not completed
- `turnOffMP` – turn off multiprocessing. Useful for profiling and debugging. Default **False**

Return

- `resultArray` – the array of results

Usage

```
from fundamentals import multiprocess
# DEFINE AN INPUT ARRAY
inputArray = range(10000)
results = multiprocess(log=log, function=functionName, poolSize=10, timeout=300,
                       inputArray=inputArray, otherFunctionKeyword="cheese")
```

3.3.10 console_logger (*function*)

console_logger(*level='WARNING'*)
Setup and return a console logger

Key Arguments

- `level` – the level of logging required

Return

- `logger` – the console logger

Usage

```
from fundamentals import logs
log = logs.console_logger(
    level="DEBUG"
)
log.debug("Testing console logger")
```

3.3.11 setup_dryx_logging (*function*)

setup_dryx_logging(*yaml_file*)
setup dryx style python logging

Key Arguments

- `level` – the level of logging required

Return

- `logger` – the console logger

Usage

```
from fundamentals import logs
log = logs.setup_dryx_logging(
    yaml_file="/Users/Dave/.config/fundamentals/fundamentals.yaml"
)
log.error("test error")
```

Here is an example of the settings in the yaml file:

```
version: 1

logging settings:
    formatters:
        file_style:
            format: '* %(asctime)s - %(name)s - %(levelname)s (%(pathname)s >
↪%(funcName)s > %(lineno)d) - %(message)s  '
            datefmt: '%Y/%m/%d %H:%M:%S'
        console_style:
            format: '* %(asctime)s - %(levelname)s: %(pathname)s:%(funcName)s:
↪%(lineno)d > %(message)s'
            datefmt: '%H:%M:%S'
        html_style:
            format: '<div id="row" class="%(levelname)s"><span class="date">
↪%(asctime)s</span>   <span class="label">file:</span><span class="filename">
↪%(filename)s</span>   <span class="label">method:</span><span class="funcName">
↪%(funcName)s</span>   <span class="label">line#:</span><span class="lineno">
↪%(lineno)d</span> <span class="pathname">%(pathname)s</span>  <div class="right
↪"><span class="message">%(message)s</span><span class="levelname">%(levelname)s
↪</span></div></div>'
```

(continues on next page)

(continued from previous page)

```
            datefmt: '%Y-%m-%d <span class= "time">%H:%M <span class= "seconds">
↪%Ss</span></span>'
    handlers:
        console:
            class: logging.StreamHandler
            level: DEBUG
            formatter: console_style
            stream: ext://sys.stdout
        file:
            class: logging.handlers.GroupWriteRotatingFileHandler
            level: WARNING
            formatter: file_style
            filename: /Users/Dave/.config/fundamentals/fundamentals.log
            mode: w+
            maxBytes: 102400
            backupCount: 1
    root:
        level: WARNING
        handlers: [file,console]
```

3.3.12 convert_dictionary_to_mysql_table (*function*)

convert_dictionary_to_mysql_table(*log*, *dictionary*, *dbTableName*, *uniqueKeyList=[]*, *dbConn=False*, *createHelperTables=False*, *dateModified=False*, *returnInsertOnly=False*, *replace=False*, *batchInserts=True*, *reDatetime=False*, *skipChecks=False*, *dateCreated=True*)

convert dictionary to mysql table

Key Arguments

- `log` – logger
- `dictionary` – python dictionary
- `dbConn` – the db connection
- `dbTableName` – name of the table you wish to add the data to (or create if it does not exist)
- `uniqueKeyList` - a lists column names that need combined to create the primary key
- `createHelperTables` – create some helper tables with the main table, detailing original keywords etc
- `returnInsertOnly` – returns only the insert command (does not execute it)
- `dateModified` – add a modification date and updated flag to the mysql table
- `replace` – use replace instead of mysql insert statements (useful when updates are required)
- `batchInserts` – if returning insert statements return separate insert commands and value tuples
  - `reDatetime` – compiled regular expression matching datetime (passing this in cuts down on execution time as it doesn't have to be recompiled everytime during multiple iterations of `convert_dictionary_to_mysql_table`)
  - `skipChecks` – skip reliability checks. Less robust but a little faster.
  - `dateCreated` – add a timestamp for dateCreated?

Return

- `returnInsertOnly` – the insert statement if requested

Usage

To add a python dictionary to a database table, creating the table and/or columns if they don't yet exist:

```
from fundamentals.mysql import convert_dictionary_to_mysql_table
dictionary = {"a newKey": "cool", "and another": "super cool",
          "uniquekey1": "cheese", "uniqueKey2": "burgers"}

convert_dictionary_to_mysql_table(
    dbConn=dbConn,
    log=log,
    dictionary=dictionary,
    dbTableName="testing_table",
    uniqueKeyList=["uniquekey1", "uniqueKey2"],
    dateModified=False,
    returnInsertOnly=False,
    replace=True
)
```

Or just return the insert statement with a list of value tuples, i.e. do not execute the command on the database:

```
insertCommand, valueTuple = convert_dictionary_to_mysql_table(
    dbConn=dbConn,
    log=log,
    dictionary=dictionary,
    dbTableName="testing_table",
    uniqueKeyList=["uniquekey1", "uniqueKey2"],
    dateModified=False,
    returnInsertOnly=True,
    replace=False,
    batchInserts=True
)

print(insertCommand, valueTuple)

# OUT: 'INSERT IGNORE INTO `testing_table`
# (a_newKey,and_another,dateCreated,uniqueKey2,uniquekey1) VALUES
# (%s, %s, %s, %s, %s)', ('cool', 'super cool',
# '2016-06-21T12:08:59', 'burgers', 'cheese')
```

You can also return a list of single insert statements using `batchInserts = False`. Using `replace = True` will also add instructions about how to replace duplicate entries in the database table if found:

```
inserts = convert_dictionary_to_mysql_table(
    dbConn=dbConn,
    log=log,
    dictionary=dictionary,
    dbTableName="testing_table",
    uniqueKeyList=["uniquekey1", "uniqueKey2"],
    dateModified=False,
    returnInsertOnly=True,
    replace=True,
    batchInserts=False
)

print(inserts)
```

(continues on next page)

(continued from previous page)

```
# OUT: INSERT INTO `testing_table` (a_newKey,and_another,dateCreated,uniqueKey2,
↪uniquekey1)
# VALUES ("cool" ,"super cool" ,"2016-09-14T13:12:08" ,"burgers" ,"cheese")
# ON DUPLICATE KEY UPDATE  a_newKey="cool", and_another="super
# cool", dateCreated="2016-09-14T13:12:08", uniqueKey2="burgers",
# uniquekey1="cheese"
```

3.3.13 directory_script_runner (*function*)

directory_script_runner(*log*, *pathToScriptDirectory*, *dbConn=False*, *waitForResult=True*, *successRule=None*, *failureRule=None*, *loginPath=False*, *databaseName=False*, *force=True*)

A function to run all mysql scripts in a given directory (in a modified date order, oldest first) and then act on the script files in accordance with the succcess or failure of their execution

The function can be run with either with an established database connection (`dbConn`) or with a mysql generated `login-path` name (`loginPath`).

with `dbConn`

Simply pass the connection `dbConn` established elsewhere in your code.

with `loginPath`

As it's insecure to pass in mysql database credentials via the command-line, run the following command from the terminal

```
mysql_config_editor set --login-path=<uniqueLoginName> --host=localhost --user=
↪<myUsername> --password
> Enter password:
```

This will store your database credentials in an encrypted file located at '~/.mylogin.cnf'. Use `mysql_config_editor print --all` to see all of the login-paths set.

The `directory_script_runner` function can work by taking advantage of mysql's `--login-path` argument so not to require knowledge of the database credentials.

Pass the login-path name via the `loginPath` parameter to use `directory_script_runner` in this manner.

If both `dbConn` and `loginPath` parameters are given, `dbConn` will be given precedent.

Key Arguments

- `log` – logger
- `pathToScriptDirectory` – the path to the directory containing the sql script to be run
- `databaseName` – the name of the database
- `force` – force the script to run, skipping over lines with errors, Default *True*
- `loginPath` – the local-path as set with `mysql_config_editor`
- `dbConn` – the database connection
- `waitForResult` – wait for the mysql script to finish execution? If 'False' the MySQL script will run in background (do not wait for completion), or if 'delete' the script will run then delete regardless of success status. Default *True*. [True|False|delete]
- `successRule` – what to do if script succeeds. Default *None* [None|delete|subFolderName]

- `failureRule` – what to do if script fails. Default *None* [None|delete|subFolderName]

Return

- None

Usage

To run the scripts in the directroy and not act on the script file use something similar to:

```
from fundamentals.mysql import directory_script_runner
directory_script_runner(
    log=log,
    pathToScriptDirectory="/path/to/mysql_scripts",
    databaseName="imports",
    loginPath="myLoginDetails"
)
```

To delete successful scripts and archive failed scripts for later inspection:

```
from fundamentals.mysql import directory_script_runner
directory_script_runner(
    log=log,
    pathToScriptDirectory="/path/to/mysql_scripts",
    databaseName="imports",
    loginPath="myLoginDetails",
    successRule="delete",
    failureRule="failed"
)
```

This creates a folder at `/path/to/mysql_scripts/failed` and moves the failed scripts into that folder.

Finally to execute the scripts within a directory but not wait for the results to return (much fast but you lose error checking in the MySQL scripts):

```
from fundamentals.mysql import directory_script_runner
directory_script_runner(
    log=log,
    pathToScriptDirectory="/path/to/mysql_scripts",
    databaseName="imports",
    loginPath="myLoginDetails",
    waitForResults=False
)
```

Setting `waitForResults` = 'delete' will trash the script once it has run (or failed . . . be very careful!)

3.3.14 get_database_table_column_names (*function*)

get_database_table_column_names(*dbConn*, *log*, *dbTable*)

get database table column names

Key Arguments

- `dbConn` – mysql database connection
- `log` – logger
- `dbTable` – database tablename

Return

- `columnNames` – table column names

Usage

To get the column names of a table in a given database:

```
from fundamentals.mysql import get_database_table_column_names
columnNames = get_database_table_column_names(
    dbConn=dbConn,
    log=log,
    dbTable="test_table"
)
```

3.3.15 insert_list_of_dictionaries_into_database_tables (*function*)

insert_list_of_dictionaries_into_database_tables(*dbConn*, *log*, *dictList*, *dbTableName*, *uniqueKeyList=[]*, *dateModified=False*, *dateCreated=True*, *batchSize=2500*, *replace=False*, *dbSettings=False*)

insert list of dictionaries into database tables

Key Arguments

- `dbConn` – mysql database connection
- `log` – logger
- `dictList` – list of python dictionaries to add to the database table
- `dbTableName` – name of the database table
- `uniqueKeyList` – a list of column names to append as a unique constraint on the database
- `dateModified` – add the modification date as a column in the database
- `dateCreated` – add the created date as a column in the database
- `batchSize` – batch the insert commands into *batchSize* batches
- `replace` – repalce row if a duplicate is found
- `dbSettings` – pass in the database settings so multiprocessing can establish one connection per process (might not be faster)

Return

- None

Usage

```
from fundamentals.mysql import insert_list_of_dictionaries_into_database_tables
insert_list_of_dictionaries_into_database_tables(
    dbConn=dbConn,
    log=log,
    dictList=dictList,
    dbTableName="test_insert_many",
    uniqueKeyList=["col1", "col3"],
    dateModified=False,
    batchSize=2500
)
```

3.3.16 readquery (*function*)

readquery(*sqlQuery*, *dbConn*, *log*, *quiet=False*)

Given a mysql query, read the data from the database and return the results as a list of dictionaries (database rows)

Key Arguments

- `log` – the logger.
- `sqlQuery` – the MySQL command to execute
- `dbConn` – the db connection
- `quiet` – ignore mysql warnings and errors and move on. Be careful when setting this to true - damaging errors can easily be missed. Default *False*.

Return

- `rows` – the rows returned by the sql query

Usage

```
from fundamentals.mysql import readquery
rows = readquery(
    log=log,
    sqlQuery=sqlQuery,
    dbConn=dbConn,
    quiet=False
)
```

3.3.17 setup_database_connection (*function*)

setup_database_connection(*pathToYamlFile*)

Start a database connection using settings in yaml file

Given the location of a YAML dictionary containing database credientials, this function will setup and return the connection*

Key Arguments

- `pathToYamlFile` – path to the YAML dictionary.

Return

- `dbConn` – connection to the MySQL database.

Usage

The settings file should be in this form, with all keyword values set:

```
db: unit_tests
host: localhost
user: utuser
password: utpass
```

And here's how to generate the connection object:

```
from fundamentals.mysql import setup_database_connection
dbConn = setup_database_connection(
    pathToYamlFile=pathToMyYamlFile
)
```

3.3.18 table_exists (*function*)

table_exists(*dbConn*, *log*, *dbTableName*)
Probe a database to determine if a given table exists

Key Arguments

- `dbConn` – mysql database connection
- `log` – logger
- `dbTableName` – the database tablename

Return

- `tableExists` – True or False

Usage

To test if a table exists in a database:

```
from fundamentals.mysql import table_exists
exists = table_exists(
    dbConn=dbConn,
    log=log,
    dbTableName="stupid_named_table"
)

print exists

# OUTPUT: False
```

3.3.19 writequery (*function*)

writequery(*log*, *sqlQuery*, *dbConn*, *Force=False*, *manyValueList=False*)
Execute a MySQL write command given a sql query

Key Arguments

- `sqlQuery` – the MySQL command to execute
- `dbConn` – the db connection
- `Force` – do not exit code if error occurs, move onto the next command
- `manyValueList` – a list of value tuples if executing more than one insert

Return

- `message` – error/warning message

Usage

Here's an example of how to create a table using the database connection passed to the function:

```
from fundamentals.mysql import writequery
sqlQuery = "CREATE TABLE `testing_table` (`id` INT NOT NULL, PRIMARY KEY (`id`))"
message = writequery(
    log=log,
    sqlQuery=sqlQuery,
    dbConn=dbConn,
    Force=False,
```

(continues on next page)

(continued from previous page)

```
    manyValueList=False
)
```

Here's a many value insert example:

```
from fundamentals.mysql import writequery
sqlQuery = "INSERT INTO testing_table (id) values (%s)"
message = writequery(
    log=log,
    sqlQuery=sqlQuery,
    dbConn=dbConn,
    Force=False,
    manyValueList=[(1,), (2,), (3,), (4,), (5,), (6,), (7,),
                   (8,), (9,), (10,), (11,), (12,), ]
)
```

3.3.20 rolling_window_sigma_clip (*function*)

rolling_window_sigma_clip(*log*, *array*, *clippingSigma*, *windowSize*)

given a sorted list of values, median sigma-clip values based on a window of values either side of each value (rolling window) and return the array mask

Key Arguments:

- `log` – logger
- `array` – the array to clean up (arrays of length < 5 are not clipped but returned unmasked)
- `clippingSigma` – the minimum sigma to clip (using median-absolute distribution as sigma proxy)
- `windowSize` – the size of the window to use when calculating the median distribution (window of 11 will use 5 values each side of the value in question)

Usage:

```
from fundamentals.stats import rolling_window_sigma_clip
arrayMask = rolling_window_sigma_clip(
    log=self.log,
    array=myArray,
    clippingSigma=2.2,
    windowSize=11)

## JUST KEEP UNMASKED VALUES
try:
    myArray = [e for e, m in zip(
        myArray, arrayMask) if m == False]
except:
    myArray = []
```

3.3.21 calculate_time_difference (*function*)

calculate_time_difference(*startDate*, *endDate*)
Return the time difference between two dates as a string

Key Arguments

- `startDate` – the first date in YYYY-MM-DDTHH:MM:SS format
- `endDate` – the final date YYYY-MM-DDTHH:MM:SS format

Return

- `relTime` – the difference between the two dates in Y,M,D,h,m,s (string)

Usage

```
from fundamentals import times
diff = times.calculate_time_difference(startDate="2015-10-13 10:02:12", endDate=
↪"2017-11-04 16:47:05")
print diff

# OUT: 2yrs 22dys 6h 44m 53s
```

3.3.22 datetime_relative_to_now (*function*)

datetime_relative_to_now(*date*)
convert date to a relative datetime (e.g. +15m, +2hr, +1w)

Key Arguments

- `date` – absolute date

Return

- a relative date

Usage

```
from fundamentals import times
relTime = times.datetime_relative_to_now(date)
```

3.3.23 get_now_sql_datetime (*function*)

get_now_sql_datetime()
A datetime stamp in MySQL format: 'YYYY-MM-DDTHH:MM:SS'

Return

- `now` – current time and date in MySQL format

Usage

```
from fundamentals import times
now = times.get_now_sql_datetime()
print now

# OUT: 2016-03-18T11:08:23
```

3.4 A-Z Index

Modules

fundamentals.commonutils	*Common tools used thoughout the fundamentals package*
fundamentals.download	*Tools used to perform download of files and HTML pages from the web*
fundamentals.files	*Tools for working with files and folders*
fundamentals.mysql	*Some handy mysql database query and insertion tools*
fundamentals.nose2_plugins	*plugins for nose2 unit testing*
fundamentals.renderer	*Render python objects as various list and markup formats *
fundamentals.stats	*Some reusable statistic functions*
fundamentals.logs	*Logger setup for python projects*
fundamentals.times	*Some time functions to be used with logging etc*

Classes

fundamentals.daemonise	A class to daemonise a python code
fundamentals.files.fileChunker	*The fileChunker iterator - iterate over large line-based files to reduce memory footprint*
fundamentals.logs.GroupWriteRotatingFileHandler	*rotating file handler for logging*
fundamentals.logs.emptyLogger	*A fake logger object so user can set ``log=False`` if required*
fundamentals.mysql.database	*a database object that can setup up a ssh tunnel (optional) and a database connection*
fundamentals.mysql.sqlite2mysql	*Take a sqlite database file and copy the tables within it to a MySQL database*
fundamentals.mysql.yaml_to_database	*Take key-values from yaml files including a table-name(s) and add them to a mysql database table*
fundamentals.renderer.list_of_dictionaries	*The dataset object is a list of python dictionaries.*
fundamentals.tools	*common setup methods & attributes of the main function in cl-util*
fundamentals.utKit	*Default setup for fundamentals style unit-testing workflow (all tests base on nose module)*

Functions

fundamentals.commonutils.getpackagepath	*Get the root path for this python package*
fundamentals.download.append_now_datestamp_to_filename	*append the current datestamp to the end of the filename (before the extension).*
fundamentals.download.extract_filename_from_url	*get the filename from a URL.*
fundamentals.download.get_now_datetime_filestamp	*A datetime stamp to be appended to the end of filenames: 'YYYYMMDDtHHMMSS'*

continues on next page

Table 30 – continued from previous page

fundamentals.download.multiobject_download	*get multiple url documents and place them in specified download directory/directories*
fundamentals.files.list_of_dictionaries_to_mysql_inserts	Convert a python list of dictionaries to pretty csv output
fundamentals.files.recursive_directory_listing	*list directory contents recursively.*
fundamentals.files.tag	Add tags and ratings to your macOS files and folders
fundamentals.fmultiprocess	multiprocess pool
fundamentals.logs.console_logger	*Setup and return a console logger*
fundamentals.logs.setup_dryx_logging	*setup dryx style python logging*
fundamentals.mysql.convert_dictionary_to_mysql_table	convert dictionary to mysql table
fundamentals.mysql.directory_script_runner	A function to run all mysql scripts in a given directory (in a modified date order, oldest first) and then act on the script files in accordance with the succcess or failure of their execution
fundamentals.mysql.get_database_table_column_names	get database table column names
fundamentals.mysql.insert_list_of_dictionaries_into_database_tables	insert list of dictionaries into database tables
fundamentals.mysql.readquery	Given a mysql query, read the data from the database and return the results as a list of dictionaries (database rows)
fundamentals.mysql.setup_database_connection	*Start a database connection using settings in yaml file*
fundamentals.mysql.table_exists	*Probe a database to determine if a given table exists*
fundamentals.mysql.writequery	*Execute a MySQL write command given a sql query*
fundamentals.stats.rolling_window_sigma_clip	*given a sorted list of values, median sigma-clip values based on a window of values either side of each value (rolling window) and return the array mask*
fundamentals.times.calculate_time_difference	*Return the time difference between two dates as a string*
fundamentals.times.datetime_relative_to_now	*convert date to a relative datetime (e.g.
fundamentals.times.get_now_sql_datetime	*A datetime stamp in MySQL format: 'YYYY-MM-DDTHH:MM:SS'*

CHAPTER

FOUR

RELEASE NOTES

v2.4.1 - May 24, 2023

- **FIXED** small bug fixes to daemonise code

v2.4.0 - May 24, 2023

- **FEATURE** added a class to help daemonise code

v2.3.12 - May 10, 2022

- **FIXED** doc fixes

v2.3.11 - March 17, 2022

- **FIXED**: deadlocked connections now attempt to reconnect.

v2.3.10 - March 7, 2022

- **ENHANCEMENT:** can now turn off multiprocessing with the `turnOffMP` parameter of `fmultiprocessing`. Needed for full profiling of code.

v2.3.9 - November 8, 2021

- **FIXED:** moved depreciated calls to yaml `load` to `safe_load`

v2.3.8 - September 29, 2021

- **ENHANCEMENT:** fundamentals is now on conda-forge

v2.3.7 - September 27, 2021

- **ENHANCEMENT:** some speed improvements in multi-downloads

v2.3.6 - August 16, 2021

- **FIXED:** no longer crashing for scripts where no settings file is passed in via CL arguments
- **FIXED:** database credentials can now be passed to the command-line again

v2.3.5 - July 30, 2021

- **FEATURE:** code bases using fundamentals can now include a 'advanced_settings.yaml' file at the root of the project which will be read before the user settings file. User settings trump settings in this 'advanced_settings.yaml' file. The purpose is to be able to have hidden/development settings.

v2.3.4 - March 16, 2021

- **ENHANCEMENT:** added loop to reattempt timed-out queries (up to 60 times)

v2.3.2 - February 23, 2021

- **FIXED:** Logger was being set from default settings file even if a custom settings file given from command line

v2.3.1 - January 6, 2021

- **FIXED:** astropy import causing grief with other package installs. Move to with function instead of module level import.

v2.3.0 - January 1, 2021

- **FEATURE:** added a stats subpackage with a `rolling_window_sigma_clip` function

v2.2.9 - December 3, 2020

- **FIXED:** relative time reporting (python 2>3ism)

v2.2.8 - November 12, 2020

- **fixed:** logging levels

v2.2.7 - November 10, 2020

- **fixed:** mysql port connection issue (with MaxScale proxy)

v2.2.6 - November 9, 2020

- **fixed:** syntax error

v2.2.5 - November 2, 2020

- **enhancement:** adding colour to logs
- **enhancement:** addition of port in database connection settings
- **fixed:** replacing depreciated 'is" syntax with ==

v2.2.4 - May 28, 2020

- **enhancement:** allowing '~' as home directory for filepaths in all settings file parameters - will be converted when initially read

v2.2.3 - May 26, 2020

- **fixed:** delimiters catered for in sql scripts

v2.2.2 - May 25, 2020

- **refactor:**`list_of_dictionaries` now returns bytes decoded into UTF-8 string when rendered to other mark-up flavour.
- **refactor:** moved module level numpy imports so that packages with fundamentals as a dependency do not have numpy as a needless dependency

v2.2.1 - May 13, 2020

- **fixed:**`directory_script_runner` function `databaseName` parameter changed to be optional

v2.2.0 - May 13, 2020

- **feature:** new `execute_mysql_script` function that allows execution of a sql script directly from file
- **refactor:** added a dbConn to the directory script runner

v2.1.7 - May 4, 2020

- **fixed:** inspect module depreciation warnings stopped
- **fixed:** MySQL error messages printed to stdout (previously hidden)

v2.1.3 - April 17, 2020

- **enhancement:** cleaned up docs theme and structure. More documentation to come.

PYTHON MODULE INDEX

c

fundamentals.commonutils, 9

d

fundamentals.download, 9

f

fundamentals.files, 10

l

fundamentals.logs, 12

m

fundamentals.mysql, 10

n

fundamentals.nose2_plugins, 11

r

fundamentals.renderer, 11

s

fundamentals.stats, 11

t

fundamentals.times, 12

INDEX

A

acquire() (*GroupWriteRotatingFileHandler method*), 16
action() (*daemonise method*), 14
add_yaml_file_content_to_database() (*yaml_to_database method*), 21
addFilter() (*GroupWriteRotatingFileHandler method*), 16
append_now_datestamp_to_filename() (*in module fundamentals.download*), 31

C

calculate_time_difference() (*in module fundamentals.times*), 45
cleanup() (*daemonise method*), 14
close() (*GroupWriteRotatingFileHandler method*), 16
connect() (*database method*), 19
console_logger() (*in module fundamentals.logs*), 36
convert_dictionary_to_mysql_table() (*in module fundamentals.mysql*), 37
convert_sqlite_to_mysql() (*sqlite2mysql method*), 20
createLock() (*GroupWriteRotatingFileHandler method*), 16
csv() (*list_of_dictionaries method*), 23

D

daemonise (*class in fundamentals*), 13
database (*class in fundamentals.mysql*), 18
datetime_relative_to_now() (*in module fundamentals.times*), 45
directory_script_runner() (*in module fundamentals.mysql*), 39
doRollover() (*GroupWriteRotatingFileHandler method*), 16

E

emit() (*GroupWriteRotatingFileHandler method*), 16
emptyLogger (*class in fundamentals.logs*), 18
extract_filename_from_url() (*in module fundamentals.download*), 31

F

fileChunker (*class in fundamentals.files*), 15
filter() (*GroupWriteRotatingFileHandler method*), 16
flush() (*GroupWriteRotatingFileHandler method*), 16
fmultiprocess() (*in module fundamentals*), 35
format() (*GroupWriteRotatingFileHandler method*), 16
fundamentals.commonutils
    module, 9
fundamentals.download
    module, 9
fundamentals.files
    module, 10
fundamentals.logs
    module, 12
fundamentals.mysql
    module, 10
fundamentals.nose2_plugins
    module, 11
fundamentals.renderer
    module, 11
fundamentals.stats
    module, 11
fundamentals.times
    module, 12

G

get_database_table_column_names() (*in module fundamentals.mysql*), 40
get_now_datetime_filestamp() (*in module fundamentals.download*), 32
get_now_sql_datetime() (*in module fundamentals.times*), 45
get_project_root() (*utKit method*), 29
getpackagepath() (*in module fundamentals.commonutils*), 31
GroupWriteRotatingFileHandler (*class in fundamentals.logs*), 15

H

handle() (*GroupWriteRotatingFileHandler method*),

17
handleError() (*GroupWriteRotatingFileHandler method*), 17

I

ingest() (*yaml_to_database method*), 21
insert_list_of_dictionaries_into_database_tables() (*in module fundamentals.mysql*), 41

J

json() (*list_of_dictionaries method*), 23

L

list() (*list_of_dictionaries property*), 26
list_of_dictionaries (*class in fundamentals.renderer*), 22
list_of_dictionaries_to_mysql_inserts() (*in module fundamentals.files*), 33

M

markdown() (*list_of_dictionaries method*), 24
module
    fundamentals.commonutils, 9
    fundamentals.download, 9
    fundamentals.files, 10
    fundamentals.logs, 12
    fundamentals.mysql, 10
    fundamentals.nose2_plugins, 11
    fundamentals.renderer, 11
    fundamentals.stats, 11
    fundamentals.times, 12
multiobject_download() (*in module fundamentals.download*), 32
mysql() (*list_of_dictionaries method*), 24

R

readquery() (*in module fundamentals.mysql*), 42
recursive_directory_listing() (*in module fundamentals.files*), 34
refresh_database() (*utKit method*), 29
release() (*GroupWriteRotatingFileHandler method*), 17
removeFilter() (*GroupWriteRotatingFileHandler method*), 17
reST() (*list_of_dictionaries method*), 25
restart() (*daemonise method*), 14
rolling_window_sigma_clip() (*in module fundamentals.stats*), 44
rotate() (*GroupWriteRotatingFileHandler method*), 17
rotation_filename() (*GroupWriteRotatingFileHandler method*), 17

S

setFormatter() (*GroupWriteRotatingFileHandler method*), 17
setLevel() (*GroupWriteRotatingFileHandler method*), 17
setStream() (*GroupWriteRotatingFileHandler method*), 17
setup() (*tools method*), 29
setup_database_connection() (*in module fundamentals.mysql*), 42
setup_dryx_logging() (*in module fundamentals.logs*), 36
setupModule() (*utKit method*), 29
shouldRollover() (*GroupWriteRotatingFileHandler method*), 17
sqlite2mysql (*class in fundamentals.mysql*), 19
start() (*daemonise method*), 14
status() (*daemonise method*), 14
stop() (*daemonise method*), 14

T

table() (*list_of_dictionaries method*), 25
table_exists() (*in module fundamentals.mysql*), 43
tag() (*in module fundamentals.files*), 34
tearDownModule() (*utKit method*), 30
tools (*class in fundamentals*), 27

U

utKit (*class in fundamentals*), 29

W

writequery() (*in module fundamentals.mysql*), 43

Y

yaml() (*list_of_dictionaries method*), 26
yaml_to_database (*class in fundamentals.mysql*), 20