libera_utils

Release 2.3.1

Libera SDC Team

Oct 15, 2024

CONTENTS:

Version: 2.3.1

CHAPTER

ONE

USER DOCS

This section is for developers including Libera Utils in their code. If you are a Libera L2 algorithm developer, you are in the right place.

1.1 Basic Usage

1.1.1 Command Line Interface

The CLI is installed as an executable in your virtual environment during installation of `libera_utils`.

Top Level Command `libera-utils`

This is the top level command that contains all the nested sub-commands. You can display the version or help text directly from this top level command.

```
libera-utils [--version] [-h]
```

Sub-Command `ecr-upload`

This is a tool to upload a docker image to AWS ECR. The image name and tag are required as arguments. The algorithm name is optional.

```
libera-utils ecr-upload [-h] image_name image_tag algorithm_name [--verbose]
```

Example usage:

```
libera-utils ecr-upload my_l2_ssw_toa_docker_image latest l2_ssw_toa
```

For all specific algorithm names to use in this command, check the *AWS constants API here* module.

Sub-Command `make-kernel jpss-spk`

```
libera-utils make-kernel jpss-spk [-h] [--outdir OUTDIR] [--overwrite] packet_data_
↪filepaths [packet_data_filepaths ...]
```

Sub-Command `make-kernel jpss-ck`

```
libera-utils make-kernel jpss-ck [-h] [--outdir OUTDIR] [--overwrite] packet_data_
↪filepaths [packet_data_filepaths ...]
```

Sub-Command `make-kernel azel-ck`

Not yet implemented

```
libera-utils make-kernel azel-ck [-h]
```

1.2 File Handling

See the smart_open API documentation here Also see *Working with NetCDF4 Files*

The libera-utils smart_open function has the capability to read and write files to/from a local directory or S3 bucket transparently. It supports a context manager pattern [HQC: "context manager pattern" may be confusing to our L2 dev users] and the usual modes for reading/writing/binary provided by most Python filelike objects:

```
from libera_utils.io.smart_open import smart_open
from libera_utils.io.filenaming import LiberaDataProductFilename
f = LiberaDataProductFilename("s3://some-bucket/LIBERA_L1B_RAD_V1-2-3_20250102T120000_
↪20250103T120000_R25005112233.nc")
with smart_open(f.path, "r") as filehandler:
    # Work with file contents
    pass
```

1.3 File Naming

The Libera Utils `Filename` classes allow reliable file naming, checking, and path management to support conformity with the Libera filenaming conventions. Each type of filename contains regex that validates every definition or update of the internally tracked filename string. These classes transparently support both S3 paths and local filepaths, including dynamic switching between the two, to simplify the transition between local development environments and AWS.

Full specifics including all available file naming classes are available *in the filenaming API documentation here*

Below is an example test using a `LiberaDataProductFilename` instance to manage a filename string, including switching between S3 and local paths to show the flexibility of the classes.

```
from pathlib import Path
from cloudpathlib import S3Path
from libera_utils.io import filenaming

```

(continues on next page)

(continued from previous page)

```
p = filenaming.LiberaDataProductFilename(
    'LIBERA_L2_CLOUD-FRACTION_V1-2-3_20270102T112233_20270102T122233_R27002112233.nc')
# Add an S3 prefix
p.path = S3Path('s3://bucket') / p.path
assert isinstance(p.path, S3Path)
# Change prefix to local
p.path = Path('/tmp/path') / p.path.name
assert isinstance(p.path, Path)
# Remove basepath altogether
p.path = p.path.name
assert isinstance(p.path, Path)
# Check that providing a bad value for a basepath doesn't pollute the instance's valid␣
↪path
try:
    p.path = '/bad/prefix' + p.path.name  # The missing / will make this fail regex␣
↪validation
    raise Exception('The previous line should have raised a ValueError')
except ValueError as e:
    assert "failed validation against regex pattern" in str(e)
assert p.path.name == 'LIBERA_L2_CLOUD-FRACTION_V1-2-3_20270102T112233_20270102T122233_
↪R27002112233.nc'
```

1.4 Working with NetCDF4 Files

NetCDF4 is a complete redesign of the NetCDF file format based on HDF5 data structures. i.e. all NetCDF4 files are HDF5 files with some additional requirements and limitation of functionality. Note that not all HDF5 files are NetCDF4 files. For more information on NetCDF5 and the underlying HDF5 structures, see the documentation here. There are several python packages (and libraries in other languages) that support reading and writing NetCDF4 files. The SDC is using the Xarray python library.

The official documentation for Xarray is here.. It includes a much more comprehensive user guide with code examples.

Xarray builds on the numpy package, introducing labels for multidimensional arrays in python. These labels come in the form of coordinates, dimensions, and attributes. Xarray is broken into two main data structures: DataArrays and DataSets. DataArrays are contained within DataSets, such that a single DataSet can hold multiple DataArrays. DataSets can then be written to NetCDF4 files.

1.4.1 Reading NetCDF4 Files

To read NetCDF4 files we can use Xarray as well. NetCDF4 files have similar structure to HDF5 files. NetCDF4 DataSets can have DataSets nested within one another. Here is an example of how to access each DataSet/Group.

```
import xarray

with xarray.open_dataset("filename", group='/') as ds:
    print(ds) # print the highest level group
```

1.4.2 Creating and Using DataArrays

See documentation on `DataArray.to_netcdf` here.

DataArrays are arrays that can handle multiple dimensions with named or labeled axes. These DataArray objects add metadata such as dimension names, coordinates, and attributes. DataArrays can be created from numpy arrays, numpy-like arrays, pandas Series, and pandas DataFrames.

```
import xarray as xr
import numpy as np
import pandas as pd

# Create the coordinates
time = pd.date_range(start='2022-01-01', periods=10, freq='D')  # 10 daily time steps
lat = np.linspace(-90, 90, 5)  # 5 latitude points from -90 to 90
lon = np.linspace(-180, 180, 8)  # 8 longitude points from -180 to 180

# Create random data
data = np.random.random((len(time), len(lat), len(lon)))

# Create the DataArray
data_array = xr.DataArray(data,
                          coords=[time, lat, lon],
                          dims=['time', 'lat', 'lon'])

# Display the DataArray
print(data_array)


print(data_array.values)  # the data in the object
print(data_array.dims)  # access the dimensions
print(data_array.coords)  # access the coors attribute
print(data_array.attrs)  # access metadata about the DataArray
```

You can create DataArrays across more dimensions as well. The number of variables in `dims` and `coords` should be equal for multiple dimensions. You can also modify the DataArrays values with scalars.

```
data_array.values = data_array.values * 2 # multiply the entire array by 2
```

1.4.3 Writing DataSets as NetCDF4

See documentation on `Dataset.to_netcdf` here.

Creating DataSets is similar to creating DataArrays, provide the data variables themselves, along with the coords, dims and attributes you want to include. The data variables should be a dictionary with each key being the name of the data and each value can be a DataSet, pandas dataframe or numpy array. Coords and Dims should be a dictionary as well.

For this example, we are creating the DataArrays first ,then will add them all into one DataSet and write that to a NetCDF4 file. When creating DataSets, Xarray will often infer the dims from the coords and data variables given, if you do not pass any while creating DataSets. When writing to the NetCDF4 files if different variables "lie" on different dimensions, they will smash them together and replace the extra values (when viewed in a file viewer) with zeros. When writing a file using Xarray, using the engine "h5netcdf" will write the file faster.

```
import random
import pandas
import numpy
import xarray as xr

data_length = random.randint(1000,2000) # creating random vector length to simulate data

# creating multiple data fields to simulate fake data
times = pandas.date_range("2014-09-06", periods=data_length)
short_wave = numpy.random.rand(data_length)
long_wave = numpy.random.rand(data_length)
total_radiance = numpy.random.rand(data_length)
split_radiance = numpy.random.rand(data_length)

# creating the DataSets from the created fields
short_wave = xr.DataArray(short_wave, coords=[times], dims=['times'])
long_wave = xr.DataArray(long_wave, coords=[times], dims=['times'])
total_rad = xr.DataArray(total_radiance, coords=[times], dims=['times'])
split_rad = xr.DataArray(split_radiance, coords=[times], dims=['times'])

# create the DataSet
ds = xr.Dataset({
    'short_wave': short_wave,
    'long_wave': long_wave,
    'total_radiance': total_rad,
    'split_rad': split_rad
},
    coords={'times': times},
)

# write to a NetCDF4 file
ds.to_netcdf('filename', group="/", mode='a', engine='h5netcdf')
```

You can specify group structure with group keyword, similar to a filesystem path (/groups/are/paths). When writing multiple DataSets to a file or if you need to append them, use keyword "mode" with value "a" to append.

1.5 Making and Using Manifest Files

All science algorithms that run on the Libera Science Data Center system need capabilities for dealing with Manifest Files. Specifics on the usage of manifest files can be found in the *Manifest API documentation here*

The `Manifest` class is designed to handle reading, writing, and interacting with manifest files during processing. It performs such tasks as validating manifest file structure and naming conventions as well as storing the manifest contents as easily accessible python objects and providing helper methods for common tasks related to manifest file handling.

```
from libera_utils.io.manifest import Manifest

my_manifest = Manifest.from_file("s3://some-dropbox/LIBERA_INPUT_MANIFEST_
↪20270102T122233.json")
# Work with manifest file
```

1.6 Logging

High quality logging is an important part of operational processing and the Libera SDC Team has made logging setup as painless as possible, while also offering a high degree of configuration for processing algorithms. See below for a general discussion of logging principles followed by some example use cases.

See the `libera_utils.logutil` API documentation here

1.6.1 Logging vs. `print`

Printing is a valid way to log from your code. However, it is limited in a few major ways:

1. You get no logging information from library code you have pulled in as dependencies.
2. There is no easy way to automate adding context to print statements such as current function, line, module, etc.
3. Formatting is only a convention with print calls, which makes log analysis and monitoring difficult.
4. There is no easy way to control the verbosity of your print statements.
5. You can only send messages to the console (stdout/stderr).

When using the Libera Utils logging module, you get:

1. Fine-grained information from all the libraries you are using.
2. Configurable standard context added to logs such as time, severity level, module, line number, function name, etc.
3. Consistent formatting to make logs easily searchable.
4. Ability to easily turn logging on/off from one place in the code.
5. Send log messages to multiple configurable destinations (console, file, etc).

See examples of these use cases in the code examples throughout this page.

1.6.2 Logging Levels in Python

- DEBUG - Detailed information, typically of interest only when diagnosing problems.
- INFO - Confirmation that things are working as expected.
- WARNING - An indication that something unexpected happened, or indicative of some problem in the near future (e.g. ‘disk space low’). The software is still working as expected.
- ERROR - Due to a more serious problem, the software has not been able to perform some function.
- CRITICAL - A serious error, indicating that the program itself may be unable to continue running.

1.6.3 Setting Up Logging in Applications

The `libera_utils.logutil` module provides utilities for configuring logging easily for file-based, stream (stdout/stderr), and custom AWS CloudWatch logging.

Simplest Logging Setup

This example allows all messages through to the console at the specified level. This does not filter out DEBUG logs from verbose libraries.

```
"""Simplest logging setup"""
import logging
from datetime import datetime, timezone
import boto3
from botocore.exceptions import NoCredentialsError
from libera_utils.logutil import configure_task_logging

logger = logging.getLogger(__name__)

if __name__ == "__main__":
    task_id = f'processing-task-{datetime.now(tz=timezone.utc).strftime("%Y-%m-%dT%H:%M:
↪%S")}'
    configure_task_logging(task_id, console_log_level="DEBUG")

    logger.debug("test debug message")

    try:
        # The following will demonstrate why we might want to filter out debug messages
        buckets = boto3.client('s3').list_buckets()
        logger.info(buckets)
    except NoCredentialsError:
        logger.error("No credentials found")
```

produces

```
2024-04-23 07:54:37,523 INFO     [libera_utils.logutil:logutil.py:257 in configure_task_
↪logging()]: Console logging configured at level DEBUG.
2024-04-23 07:54:37,523 DEBUG    [__main__:scratch_11.py:15 in <module>()]: test debug␣
↪message
2024-04-23 07:54:37,523 DEBUG    [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from creating-client-class.iot-data to creating-client-class.iot-
↪data-plane
2024-04-23 07:54:37,524 DEBUG    [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from before-call.apigateway to before-call.api-gateway
2024-04-23 07:54:37,524 DEBUG    [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from request-created.machinelearning.Predict to request-created.
↪machine-learning.Predict
2024-04-23 07:54:37,524 DEBUG    [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from before-parameter-build.autoscaling.CreateLaunchConfiguration␣
↪to before-parameter-build.auto-scaling.CreateLaunchConfiguration
2024-04-23 07:54:37,525 DEBUG    [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from before-parameter-build.route53 to before-parameter-build.
↪route-53
```

(continues on next page)

(continued from previous page)

```
2024-04-23 07:54:37,525 DEBUG     [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from request-created.cloudsearchdomain.Search to request-created.
↪cloudsearch-domain.Search
2024-04-23 07:54:37,525 DEBUG     [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from docs.*.autoscaling.CreateLaunchConfiguration.complete-section␣
↪to docs.*.auto-scaling.CreateLaunchConfiguration.complete-section
2024-04-23 07:54:37,526 DEBUG     [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from before-parameter-build.logs.CreateExportTask to before-
↪parameter-build.cloudwatch-logs.CreateExportTask
2024-04-23 07:54:37,526 DEBUG     [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from docs.*.logs.CreateExportTask.complete-section to docs.*.
↪cloudwatch-logs.CreateExportTask.complete-section
2024-04-23 07:54:37,526 DEBUG     [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from before-parameter-build.cloudsearchdomain.Search to before-
↪parameter-build.cloudsearch-domain.Search
2024-04-23 07:54:37,526 DEBUG     [botocore.hooks:hooks.py:482 in _alias_event_name()]:␣
↪Changing event name from docs.*.cloudsearchdomain.Search.complete-section to docs.*.
↪cloudsearch-domain.Search.complete-section
...and much much more
```

Notice the huge volume of DEBUG messages originating from loggers in the `botocore` *package.*

Filtered Logging Setup

Now we want to alter the code above to take advantage of the ability to reduce the amount of DEBUG spam from libraries that we are not interested in. Note the use of `__main__` in the `limit_debug_loggers` tuple. This allows debug messages that originate at the level of a runnable python script that is wrapped in the standard `if __name__=="__main__"` guard.

```python
"""Simplest logging setup"""
import logging
from datetime import datetime, timezone
import boto3
from botocore.exceptions import NoCredentialsError
from libera_utils.logutil import configure_task_logging

logger = logging.getLogger(__name__)

if __name__ == "__main__":
    task_id = f'processing-task-{datetime.now(tz=timezone.utc).strftime("%Y-%m-%dT%H:%M:
↪%S")}'
    configure_task_logging(task_id, limit_debug_loggers=("__main__", "libera_utils"),␣
↪console_log_level="DEBUG")

    logger.debug("test debug message")

    try:
        # The following will demonstrate why we might want to filter out debug messages
        buckets = boto3.client('s3').list_buckets()
        logger.info(buckets)
    except NoCredentialsError:
        logger.error("No credentials found")
```

produces

```
2024-04-23 07:52:56,009 INFO     [libera_utils.logutil:logutil.py:257 in configure_task_
↪logging()]: Console logging configured at level DEBUG.
2024-04-23 07:52:56,009 DEBUG    [__main__:scratch_11.py:15 in <module>()]: test debug␣
↪message
2024-04-23 07:52:56,186 ERROR    [__main__:scratch_11.py:22 in <module>()]: No␣
↪credentials found
```

Contrived Runnable Example

This illustrates how the `limit_debug_loggers` kwarg works for preventing other libraries from logging at DEBUG level while still allowing DEBUG messages from libraries you care about.

```
"""Logging setup contrived example"""
from datetime import datetime, timezone
import logging
from libera_utils.logutil import configure_task_logging

task_id = f'processing-task-{datetime.now(tz=timezone.utc).strftime("%Y-%m-%dT%H:%M:%S")}
↪'
configure_task_logging(task_id, limit_debug_loggers=("my_package",), console_log_
↪level=logging.DEBUG)
# my_log is a logger from inside your library (name prefixed with your library name).
my_log = logging.getLogger('my_package.my_application')
# The following is an example. External libraries will create their own loggers␣
↪internally.
external_library_log = logging.getLogger('some_spammy_library')

my_log.debug('subtle but important message gets passed through')
external_library_log.debug('this external library debug spam gets filtered out')
external_library_log.info('and external library info messages still get through')
```

Configuring Stream Logging

Stream logging needs only a level and it defaults to `INFO`. Stream logging cannot be disabled but is easily ignored.

Note: You can change the default formatter for stream logging from plaintext to json by passing `console_log_json=True` to `configure_task_logging`. This is convenient for logging in AWS services that push their stdout logs to CloudWatch.

```
"""Example of setting up console logging (JSON formatted)"""
import logging
from libera_utils.logutil import configure_task_logging

configure_task_logging("test-task-id-1", console_log_json=True, console_log_
↪level=logging.DEBUG)
```

Configuring Filesystem Logging

Filesystem logging needs only a log directory (it always logs at DEBUG level). If you don't pass `log_dir`, no file-based logging will occur. The log directory must exist and will not be dynamically created.

Note: Logging to a directory is really only useful for local testing. Any logs written to a directory in a docker container will evaporate upon completion of the docker container process.

```
"""Example of setting up file-based logging"""
from pathlib import Path
from libera_utils.logutil import configure_task_logging

configure_task_logging("test-task-id-1", log_dir=Path("/tmp"))
```

1.6.4 Logging Exceptions

The Python logging module provides logging calls associated with each level. In addition, it provides a logging call for logging exceptions that includes the current stack trace for debugging.

```
"""Example logging calls"""
import logging

logger = logging.getLogger(__name__)

if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)
    logger.debug("test debug")
    logger.info("test info")
    logger.warning("test warning")
    logger.error("test error")
    logger.critical("test critical")

    try:
        raise ValueError("example exception to log")
    except ValueError:
        logger.exception("encountered exception")  # This logs the message followed by␣
↪the exception traceback
        # raise  # Optional re-raise of exception after logging it
```

1.6.5 Concept: Module Level Logging

(AKA library logging)

Module level logging is the practice of defining a logger at the top of each module (.py file) and using that logger object for the entire module. Modules in libera_utils should all have module level loggers when appropriate. e.g.

```
"""Example of module level logger instantiation"""
import logging

logger = logging.getLogger(__name__)

# Module code
```

One advantage of module level logging is that it provides a logger, named for the module from which it is logging but doesn't configure the logger at the module level. Since much of this code is intended to be reused by others, we avoid configuring loggers in reusable code. If a logger is needed in a context, it should be configured at the "application level". That is, the top level application (e.g. CLI tool) that is running a process should take care of logging configuration and assume that each module has generic module level loggers configured for the internal code to use.

Another advantage of module level logging is that our loggers come out with automatically structured names like `libera_utils.db.database` and they all start with `libera_utils`. This allows us to treat those loggers differently than those named, for example, `some_spammy_library.emit_spam`. In our logging setup, we allow users to turn off debug messages for all loggers that aren't named with specific prefixes. This allows us to pass up debug messages from our code but ignore debug messages from dependency code (AWS boto APIs in particular spam a LOT of debug messages).

1.6.6 Fully Customized Logging

If you want complete control over your logging configuration, you can use our provided `configure_static_logging` function, which reads a YAML configuration file that represents a Python logging configuration. This is a completely static configuration and should be supplied as part of your processing algorithm application code.

```
"""Example of configuring logging with static config file"""
import logging
from pathlib import Path
from libera_utils.logutil import configure_static_logging

config = Path("/path/to/config_file.yml")
configure_static_logging(config)

logger = logging.getLogger()
logger.info("handling depends on your supplied configuration")
```

An example of a logging config file:

```
# Example parameterized logging configuration
version: 1
disable_existing_loggers: False
formatters:
    json:
        format: '{"time": "%(asctime)s",
                  "level": "%(levelname)s",
                  "module": "%(filename)s",
                  "function": "%(funcName)s",
                  "line": %(lineno)d,
                  "message": "%(message)s"}'
    plaintext:
        format: "%(asctime)s %(levelname)-9.9s [%(filename)s:%(lineno)s in
↪%(funcName)s()]: %(message)s"
handlers:
    console:
        class: logging.StreamHandler
        formatter: plaintext
        level: INFO
        stream: ext://sys.stdout
    logfile:
```

(continues on next page)

(continued from previous page)

```
        class: logging.handlers.RotatingFileHandler
        formatter: plaintext
        level: DEBUG
        filename: /tmp/libera_utils_test_log.log
        maxBytes: 1000000
        backupCount: 3
root:
    level: INFO
    propagate: True
    handlers: [console, logfile]
loggers:
    libera_utils:
        qualname: libera_utils
        level: DEBUG
        handlers: []
```

CHAPTER

TWO

DEVELOPER DOCS

This section is for developers working on the Libera Utils package but also includes reference documentation for the tools used for development and how to use them (e.g. git and Docker).

2.1 Setting Up a Development Environment

If you have access to the internal LASP confluence space, please refer to the following resources:

- Python Development Environment Management

2.1.1 Managing Multiple Base Python Versions

In order to develop with multiple different versions of Python and create virtual environments associated with different versions of Python, you will need multiple base Python interpreters. There are several ways to manage this including Conda, PyEnv, and building Python from source. We recommended using Conda and outline the steps below for using Conda to manage multiple base Python installations.

1. Install miniconda according to the official documentation. If you already have miniconda or anaconda installed, you can skip this step.
2. Optionally run `conda config --set auto_activate_base false` to add a configuration to your `.condarc` file to disable auto-activation of the `base` conda environment on shell startup.
3. Create a conda environment with your preferred version of python: `conda create -n conda-python3.11 python=3.11`
    - Note: Name this environment with a convention that makes sense to you for a base interpreter. *Do not delete this conda environment!* Deleting it will break all subsequent virtual environments based on it.
4. The Python interpreter provided by your new conda environment is a full base interpreter and you can use it to create virtual environments. You can find the full path to the base interpreter by running something similar to the following (run `conda env list` to see why this works):

```
PATH_TO_PYTHON=$(conda env list | grep "conda-python3.11" | awk '{print $2}')/bin/
↪python
$PATH_TO_PYTHON -m venv path/to/new/venv
```

2.1.2 Installing Poetry

Poetry is a command line tool that helps manage a python development environment, including package management, virtual environment management, and package building.

Poetry official installation instructions can be found here: https://python-poetry.org/docs/#installation

To ensure that Poetry is always available and in the PATH it is recommended to install Poetry with your pre-installed system python interpreter rather than as a package in a conda environment or in a virtual environment. The specific version of python with which you install Poetry is inconsequential (as long as it is currently supported by Poetry). If your system python is not supported by Poetry, you can install Poetry in your conda base environment. Just remember that Poetry will only be available when that environment is activated. Things can get a bit confusing when you have a conda environment active and a derived virtual environment activated on top of it.

Once poetry is installed, check that it works by running `poetry --version`. You should get something like

```
Poetry version 1.4.0
```

Installing Poetry with System Python

Ensure that all your virtual environments and conda environments are deactivated and that `which python3` refers to your system python interpreter (usually `/usr/bin/python3`).

```
curl -sSL https://install.python-poetry.org | python3 -
```

Installing Poetry in the Conda `base` Environment

```
conda activate
conda install -y poetry
poetry --version
conda deactivate
poetry --version  # This should fail to find Poetry
```

2.1.3 Configuring Poetry

We recommend configuring Poetry to create virtual environments in project directories by default.

```
poetry config virtualenvs.in-project true
```

This command will write a value to a config file for your global poetry installation. On Mac this is in `~/Library/Application\ Support/pypoetry/config.toml`.

2.1.4 Setting Up Development Virtual Environment(s)

Poetry manages virtual environments for you on a per-package and per-python-version basis. However, Poetry will dynamically detect the presence of an activated virtual environment and use that if present.

1. Deactivate all Conda environments and virtual environments (except for the conda environment which contains Poetry, if applicable).
2. Save the path to the version of Python you want to use for development

```
PATH_TO_PYTHON=$(conda env list | grep "conda-python3.11" | awk '{print $2}')/bin/
↪python
```

3. Instruct Poetry to create a managed virtual environment

```
poetry env use $PATH_TO_PYTHON
```

   This will create a virtual environment. If you have enabled `virtualenvs.in-project` as described above, it will be created in your project directory in `.venv`.

4. Configure your IDE to recognize the correct poetry-managed virtual environment for the version you wish to develop with.
5. Run `poetry env info` and verify that Poetry is recognizing your virtual environment properly:

```
Virtualenv
Python:         3.9.9
Implementation: CPython
Path:           /Users/myuser/path/to/libera_utils/venv
Valid:          True
```

Changing Python Versions

It is common to recreate your virtual environment on a regular basis in order to use different python versions. You can do this with

```
poetry env use /full/path/to/python
```

Poetry will recreate your virtual environment in the `.venv` directory if `virtualenvs.in-project` is set. Otherwise it will create a virtual environment in `~/Library/Caches/pypoetry/virtualenvs`.

Installing Dependencies

1. Run `poetry install` in the same directory as the `pyproject.toml` file. You should see poetry solving the dependency tree and then installing dependencies. This also installs dev group dependencies, as specified in `pyproject.toml`. Lastly you should see it installing the local package.
2. To install optional "extra" dependencies, run `poetry install -E extra_name1 -E extra_name2`. These extra dependencies are specified in `pyproject.toml` under `[tool.poetry.extras]`. Note that any subsequent `poetry install` command without `--extras` will implicitly uninstall any previously installed extras.
3. To install dependency "groups" (think labels), which may or may not be optional, use the `--with` and `--without` flags for Poetry. e.g. `poetry install --with docgen` will install the dependencies for the optional group "docgen".

4. Verify that the `libera_utils` package was installed correctly by running `libera-utils --version`. This runs the `libera-utils` command line utility that is included in the package. This can also be run with `poetry run libera-utils --version`.
5. Next, *go run the tests*.

2.2 Configuration

Configurations are stored in a JSON file `config.json` but we allow overriding those values with environment variables. If, at any point in the code, the config is queried (`config.get(key)`) for a key that isn't present in the top level of the JSON file, a warning is issued to add a default value to the JSON file. This helps ensure that we can track all possible configuration values in the JSON config rather than having to bookkeep them elsewhere.

2.3 Testing

2.3.1 Testing Locally

Testing is run with `pytest`. To run all tests, make sure the dev dependencies are installed and navigate to the *tests* folder in the repository and run:

```
pytest
```

With coverage for generating reports on code coverage:

```
# Create coverage data (stored in .coverage)
pytest --cov=libera_utils --junit-xml=junit.xml
# Generate interactive HTML coverage report
pytest --cov-report=html:coverage_report --cov=libera_utils
# Generate Corbertura-compatible XML report
pytest --cov-report=xml:coverage.xml --cov=libera_utils
```

2.3.2 Testing in Docker

To run the unit tests in docker, run

```
docker-compose up [--build] --exit-code-from=tests tests --attach=tests
```

This ensures the dev database server is up, runs the latest flyway migrations against it, and runs the tests container service defined in the `docker-compose.yml` file. The `--build` option forces docker to rebuild the testing container image before running (e.g. if things have changed).

Copying Test Report Artifacts from Docker

When we run tests in Docker on Jenkins, we often want to copy and save Corbertura and JUnit test reports. Jenkins has facility for doing this easily with

```
always {
    junit '**/*junit.xml'
    cobertura coberturaReportFile: '**/*coverage.xml'
}
```

The challenge when running in Docker is to make these test artifacts available to Jenkins. By default these files exist only inside the Docker container so we must copy them out. Do this with

```
docker-compose --exit-code-from tests up tests
docker-compose cp tests:/path/to/report.xml .
```

2.3.3 Static Analysis

NPR7150.2C requires that we perform static analysis of our codebase to check for common vulnerabilities and statically detectable code weaknesses.

Pylint for Code Style

Pylint is a powerful and highly configurable static analysis tool that analyzes Python code for coding standards (e.g. PEP8), code smells, type errors, and violations of common best practices. Together, these violations are common code weaknesses and we strive to eliminate them.

To run pylint locally, run

`pylint libera_sdp`

from the repo root. This will lint the `libera_sdp` directory and automatically pick up our `pylintrc` configuration file.

To run pylint inside Docker, run

`docker-compose up [--build] linting`

This starts up the linting service described in `docker-compose.yml`, which runs pylint inside a testing docker container.

Bandit for Automated Security Testing (AST)

Bandit is a security vulnerability tester that analyzes Python code for common weaknesses, including the "official" CWE set provided by the MITRE Corp..

To run bandit locally, run

`bandit -r libera_sdp`

from the repo root. This will recursively (`-r`) analyze the `libera_sdp` directory and report results.

To run bandit inside Docker, run

`docker-compose up [--build] ast`

This starts up the `ast` service described in `docker-compose.yml`, which runs bandit inside a testing docker container.

Static Analysis pre-commit Git Hook

The static analysis checking can be annoying when you only catch it after you have committed, rebased, and put up a PR. To alleviate that suffering, here is a pre-commit git hook that will execute before every commit and refuse to continue until you have fixed your static analysis problems.

```
#!/bin/sh

# .git/hooks/pre-commit

# Redirect output to stderr.
exec 1>&2

# Run pylint.
echo "Running pylint..."
files_to_check=$(git diff --name-only --cached --diff-filter=AM libera_utils | grep '.py$
↪')
if [ -n "$files_to_check" ]; then
  pylint $files_to_check
else
  echo "No .py file changes to lint"
fi

# Capture the output of pylint.
pylint_exit_code=$?

# If pylint returns a non-zero exit code, cancel the commit.
if [ $pylint_exit_code -ne 0 ]; then
  echo "Pylint check failed, aborting commit..."
  exit 1
fi

# Otherwise, proceed with the commit.
echo "Pylint check passed, proceeding with commit..."

echo "Running bandit..."
bandit -r --quiet libera_utils
bandit_exit_code=$?

# If Bandit returned a non-zero exit code, cancel the commit.
if [ $bandit_exit_code -ne 0 ]; then
    echo "Bandit check failed, aborting commit..."
    exit 1
fi

# Otherwise, proceed
echo "Bandit AST passed, proceeding with commit..."

exit 0
```

2.4 Build and Release

2.4.1 Local package building for CLI testing

To build the package locally for testing especially for the cli interface, use the following steps:

1. Ensure that you have activated a virtual environment where you would like libera-utils to be installed.
2. run `python -m pip install .` from the root of the repository.
3. You should now be able to run the `libera-utils --version` command from the command line as see that the version number matches the one in *pyproject.toml*.

2.4.2 Release Process

Atlassian Git Workflow Reference:

1. Create a release candidate branch named according to the version to be released. This branch is used to polish the release while work continues on dev (towards the next release). The naming convention is `release/X.Y.Z`
2. Bump the version of the package to the version you are about to release, either manually by editing `pyproject.toml` or by running `poetry version X.Y.Z` or bumping according to a valid bump rule like `poetry version patch` (see poetry docs).
3. Open a PR to merge the release branch into main. This informs the rest of the team how the release process is progressing as you polish the release branch.
4. When you are satisfied that the release branch is ready, merge the PR into `main`. This should be a purely "fast-forward" merge. Do not delete the release branch when merging as you will need it later.
5. Check out the `main` branch, pull the merged changes, and tag the newly created merge commit with the desired version `X.Y.Z` and push the tag upstream.

```
git tag -a X.Y.Z -m "version release X.Y.Z"
git push origin X.Y.Z
```

6. Checkout the tag you just created (ensures the correct version is recorded in the build artifacts) and build the package (see below). Check that the version of the built artifacts is as you expect (should match the version git tag).
7. Optionally distribute the artifacts to PyPI/Nexus if desired (see below).
8. Open a PR to merge `release/X.Y.Z` back into `dev` so that any changes made during the release process are also captured in `dev`. This should be a purely "fast-forward" merge.

2.4.3 Building and Distribution to Public PyPI

1. Ensure that `poetry` is installed by running `poetry --version`.
2. Checkout the tag of the version you are releasing.
3. To build the distribution archives, run `poetry build`.
4. To upload the wheel to PyPI, first set your environment variables with the API token for the correct PyPI account:

```
export PYPI_USERNAME=__token__
export PYPI_TOKEN=<Your API Token>
```

Then run `poetry publish --username $PYPI_USERNAME --password $PYPI_TOKEN`. You will need the account information for the `liberasdc` PyPI account.

2.4.4 Building and Distribution to Internal LASP Nexus PyPI

Adding a Poetry Repository for the Nexus PyPI

This Poetry configuration allows Poetry to distribute to non-standard (private) package indexes.

You will need the account information for your LASP Nexus account. Note that the repository, which is named lasp-pypi in this example must first be configured according to the Poetry docs here. To configure the repo for publishing, run

```
poetry config repositories.lasp-pypi https://artifacts.pdmz.lasp.colorado.edu/repository/
↪lasp-pypi/
```

Note that the trailing slash is required at the end of the URL.

Distributing an Internal Release to Nexus

The intention is that we can have a bleeding edge local version on Nexus that is not available to the general public. This Nexus release will be based on the `dev` branch and will generally be less stable than the version released to the public PyPI.

1. Ensure that `poetry` is installed by running `poetry --version`.
2. Checkout the `dev` branch, or really whatever branch you want to release as the internal version.
3. To build the distribution archives, run `poetry build`.
4. Visit Nexus at https://artifacts.pdmz.lasp.colorado.edu/#browse/browse:lasp-pypi and remove the previous version of `libera_utils` (just delete it, this is an internal `dev` release).
5. To upload the wheel to Nexus, run

   ```
   poetry publish --repository lasp-pypi --username your-nexus-username --password␣
   ↪*********
   ```

2.5 Documentation Generation for Libera Utils

2.5.1 Creating Documentation with Sphinx

Sphinx documentation is a python library for generating documentation from docstring comments throughout a codebase in combination with other rst or md pages separate from the generated documentation. We use it to generate automatic API documentation for the codebase as well as building developer documentation and user documentation from markdown and restructured text documents (such as this one).

You will need `make` installed on your machine in order to build sphinx docs.

Compatible Comments and Docstrings

Docstrings in this project are expected in the Numpy dosctring format.

Writing Custom Documentation Pages

If you wish to add pages that are not part of the generation from the code such as reference pages, use the following steps.

- Create your files and folders and add them to the project folder `doc/source/`
- edit the `doc/source/index.rst` file to add the file you have just created to the toctree (table of contents tree)
- If you added developer documentation, instead of editing the `index.rst` edit the `doc/source/developer-docs.rst` document or create a new folder and associated rst document containing a toctree that includes your new file.

The folder structure is an organizational convention but the actual page structure comes from the toctree structure in the rst files (note that these documents could also be markdown but rst has better support for TOC listings).

Note: This project is configured to read and interpret both markdown (.md) and reStructured Text (.rst) documents. The following are two basic guides to writing proper comments that will create well formatted outputs.

- reStructured Text
- Markdown

If you have need to translate between rst and markdown, use `pandoc`.

2.5.2 Building HTML Documentation Locally

This should all be done while in the virtual environment that is configured for this project. See documentation dev-environment-setup.md in the libera-sdp repository for poetry instructions.

1. Run `poetry update --with docgen`
    - This ensures that the poetry install is up-to-date, including the "docgen" dependency group.
2. Run `poetry install --with docgen`
    - This installs all project dependencies, including the `pyproject.toml` docgen group (e.g. Sphinx, etc.)
    - This also ensures that the project itself is installed with its most recent version (as defined in `pyproject.toml`)
3. Navigate to the Sphinx document source folder `cd ./doc`
4. Build the html files in the build folder using the make command `make html`
5. Open the generated `build/index.html` file to go to the documentation homepage.

2.5.3 Automatic Documentation Publishing with ReadTheDocs

ReadTheDocs.org is a free service (ad-supported) for developers to publish documentation for open source code for free. The only requirement is that the repository is publicly available to clone. You're probably reading this on readthedocs.io.

Our configuration file for readthedocs is located in `.readthedocs.yaml`. The Libera SDC readthedocs account is shared between the Libera SDC development team and is not tied to a specific developer. Configuration for how readthedocs decides which versions to build is configured in the readthedocs account page.

2.6 Git Usage

2.6.1 Basic Workflow

This workflow is knows as GitFlow

This is the ideal order of events. If you know what you're doing, it is possible to deviate from this process in minor ways.

1. Create a branch from `dev` for a feature. Name it based on your feature or ticket. e.g. `feature/LIBSDC-XXX-add-something-cool`
2. Add commits.
3. (Optional) Put up a PR prefixed with `WIP:` to signify a work in progress while still allowing team members visibility.
4. Ensure your branch is rebased onto `dev`
5. Ensure your changes are squashed into a single commit (e.g. by running `git rebase -i dev` and resolving conflicts).
6. Put up a PR to merge into `dev`
7. Wait for review.
8. Merge using the fast-forward only strategy in Bitbucket. If properly squashed, should only add 1 commit.

If any of this doesn't make sense, don't just run commands! Ask someone! You can really hose your local repo state and even lose your work if you don't know what you're doing.

2.6.2 Reasons for Rebasing

Rebasing allows us to keep our git history completely linear and avoids creating unnecessary merge commits.

2.6.3 Reasons for Squashing

Squashing reduces the total number of commits in the repository to ~1 per pull request. Since each commit contains a full snapshot of the codebase, this drastically reduces the amount of storage necessary to host our git repo and makes life slightly easier for our beloved Web Team.

2.7 Git LFS Usage

We use Git LFS to store large files in a way that doesn't blow up the size of our repo on the git server. Usually each commit contains a snapshot of the repository at that point in time. If you store a 100MB file, your entire repo size will be 100MB * number of commits, which can easily balloon to a big number. Incidentally, this is also why we squash PRs to reduce the number of commits in the main branch over time.

Git LFS Documentation

Tutorial on Using Git LFS

Atlassian Docs on Git LFS

2.7.1 Set Up

Install Git LFS according to the Git LFS official documentation (linked above).

Run the following to initialize Git LFS for your user:

```
git lfs install
```

2.7.2 A Note on Git LFS Authentication and Git GUIs

Git LFS authenticates separately from Git. Historically, Git LFS supported only HTTP auth, which was a huge pain because best practice is to use SSH to authenticate with Git servers (note that GitHub has actually deprecated HTTP authentication). This situation meant that even if you were using SSH for git auth, you still had to provide and store HTTP credentials for Git LFS.

However, as of Git LFS v3.0, SSH authentication is supported!

Git GUIs will require you to configure authentication for both Git and Git LFS. It is recommended to use SSH for both but the configuration processes for GUI programs (and IDEs) are different so we're not addressing it here. GLHF!

2.7.3 Tracking New Files in LFS

LFS keeps track of which files are stored in LFS via the `.gitattributes` file.

To track specific files:

```
git lfs track "<pattern>"  # The double quotes matter to prevent shell expansion
# e.g. track all files in every directory named test_data
git lfs track "**/test_data/*"
# This ^ grabs all the filepaths that match and writes them directly into .gitattributes
```

To track files based on a pattern in `.gitattributes`:

```
# .gitattributes
# Track all netCDF files that live anywhere inside a test_data directory
**/test_data/**/*.nc
```

Documentation on Pattern Syntax

2.7.4 Tracking Existing Files in LFS

If you have already committed a file and you wish to move that file to Git LFS, you can:

Add it specifically using `git lfs track` e.g.

```
git lfs track my_large_file.big
```

This method appears to have some magic sugar behind it that automatically removes and re-adds the file to git tracking.

Alternatively, you can add the appropriate pattern to `.gitattributes` and then remove and re-add the file to git history so Git LFS picks it up.

```
echo "**/*.big" >> .gitattributes  # Add pattern to .gitattributes
git rm --cached my_tracked_file.big  # Remove file from git tracking
git add my_tracked_file.big  # Git LFS should pick it up at this point
```

This method is preferable if you want your files generally tracked by pattern rather than individually.

NOTE: As a bit of a trick, you can combine the above strategies by running `git lfs track` on the pattern you wish to store in Git LFS. Then replace the individual records added to `.gitattributes` with the appropriate generic pattern that matches the specific files to be tracked.

2.7.5 Useful Git LFS Commands

List all files in the current git ref (branch, commit, tag, etc.) currently managed by Git LFS:

```
git lfs ls-files
```

Update the files in your `.git/lfs` directory with the version for your current ref:

```
git lfs fetch
```

Convert local pointer files to full files (from `.git/lfs` directory):

```
git lfs checkout
```

Combine fetch and pull into one step:

```
git lfs pull
```

2.8 Docker

2.8.1 Docker Installation

Follow the Docker Desktop installation instructions here: https://docs.docker.com/desktop/mac/install/

Note: On recent Docker Desktop for Mac, the `compose` command has been added as a subcommand of the `docker` command. This is not the case on Linux distributions, where (currently) `docker-compose` is a hyphenated command and is a different executable that must be installed separately.

To install `docker-compose` on Linux, see instructions here: https://docs.docker.com/compose/install/

2.8.2 Building Docker Images

Using docker compose:

```
docker-compose build
```

2.8.3 Nexus Docker Container Registry

Basics

Nexus is an artifact storage service run by the LASP webteam. It contains an area that we use as a container registry, similar to an AWS ECR or Dockerhub. You will need to obtain WebIAM permissions from the webteam to access Nexus and its web UI is only available from inside LASP's network (e.g. via VPN).

Nexus is located in the DMZ at https://artifacts.pdmz.lasp.colorado.edu/

Usage

Fundamentally, all that the container registry does is provide a web API that the docker cli can interface with. Every docker command that interacts with a registry is just communicating with Nexus via HTTP. The documentation of this API is here (knowledge of the API spec is not required for pushing docker images): https://docs.docker.com/registry/spec/api/

The URL for the Nexus container registry is:

Internal: `docker-registry.pdmz.lasp.colorado.edu`

External: `lasp-registry.colorado.edu`

Note: At time of writing, the external registry URL is having problems due to a load balancer issue on LASP's network. It's recommended that you use the internal URL, which requires you to be on the LASP network.

Docker Login

1. Check that you are not already logged in by running `cat ~/.docker/config.json` and ensuring that neither Nexus registry URL is in the list of logged in registries.
2. Run `docker login docker-registry.pdmz.lasp.colorado.edu` (you will be prompted for your WebIAM username and password).
3. You should see a success message and your `~/.docker/config.json` file should now contain a reference to the registry url.

Pulling and Pushing

Images in a docker registry are uniquely identified by their URL path. "Repositories" within a registry are separated only by path segments and permissions. That is, there is nothing special about the `libera` directory in the registry vs `libera/mysubdir/myimage`. You can push images to any location for which you have permissions.

To push an image that was created locally as `my-image:my-tag`:

1. Re-tag the image with the registry URL by running:

```
docker tag my-image:my-tag docker-registry.pdmz.lasp.colorado.edu/libera/my-
↪image:my-tag
```

The location to which it will be pushed is entirely defined by the tag you give it.

2. Assuming you are already logged in to the registry, run:

```
docker push docker-registry.pdmz.lasp.colorado.edu/libera/my-image:my-tag
```

Note: If there is already an image at this location in the registry with the same tag (but a different SHA), the tag will be removed from the old image and it will remain there but "untagged". You can verify the association between an image manifest (uniquely identified by its SHA) and a tag by inspecting the tag in the Nexus web UI.

To pull an image that is in the Nexus registry as `my-image:my-tag`, run

```
docker pull docker-registry.pdmz.lasp.colorado.edu/libera/my-image:my-tag
```

Re-Tagging

For example, you just pushed `my-image:v1.1.1` but you also want it tagged as `latest` so it becomes the default image. To tag an image with a second tag, simply tag it locally (`docker tag ...`) and push it again. Assuming the image has not changed, it will associate the manifest with the new tag (the old tag association will also remain).

2.9 Usage of SPICE

Fun fact: SPICE stands for "Spacecraft Planet Instrument C-matrix Events," which were the original primary concerns of those developing the library.

2.9.1 Static Kernels Generated at Libera SDC

These kernels are part of the package data, are manually edited, and change rarely.

Frame Kernel (FK)

e.g. `libera_fk_v01.tf`

Contains reference frame definitions for JPSS and Libera

Spacecraft Clock Kernel (SCLK)

e.g. `jpss_sclk_v01.tsc`

Contains specification of the spacecraft clock on JPSS.

Instrument Kernel (IK)

e.g. `libera_cam_ik_v01.ti`, `libera_rad_ik_v01.ti`

Contains geometry specification data of the Libera instruments.

2.9.2 Dynamic Kernels Generated at Libera SDC

These kernels are binary generated kernels. They are ancillary data products and are created as part of pipeline processing.

JPSS Ephemeris Kernel (SPK)

e.g. `libera_jpss_20210408t235850_20210409t015849_vM3m14p159_r25365125959.bsp`

Contains ephemeris data – coordinates in ITRF93 frame – for the JPSS spacecraft body.

JPSS Attitude Kernel (CK)

e.g. `libera_jpss_20210408t235850_20210409t015849_vM3m14p159_r25365125959.bc`

Contains attitude data – quaternion rotations from J2000 – for the JPSS spacecraft body.

Azimuth Rotation Mechanism Attitude Kernel (CK)

e.g. `libera_azrot_20210408t235850_20210409t015849_vM3m14p159_r25365125959.bc`

Contains attitude data for the Libera Azimuth Rotation mechanism.

Note: there is currently no mechanism for creating this kernel because no telemetry data exists.

Elevation Scan Mechanism Attitude Kernel (CK)

e.g. `libera_elscan_20210408t235850_20210409t015849_vM3m14p159_r25365125959.bc`

Contains attitude data for the Libera Elevation Scan mechanism.

Note: there is currently no mechanism for creating this kernel because no telemetry data exists.

2.9.3 Kernels Retrieved from NAIF

These kernels are generated at NAIF. We download them as needed using the `KernelFileCache` class, which is configured with a NAIF index page URL and a regex string that finds the proper download URL on the index page. The downloaded file is put in a cache so the download only occurs after the cached data is of a specified age. If we want to effectively cache some kernels indefinitely, we can put them in an S3 bucket and retrieve them from there instead of from the NAIF website.

Leapseconds Kernel (LSK)

e.g. `naif0012.tls`

Contains leapsecond data used by time conversion routines.

Development Ephemeris Kernel (SPK)

e.g. `de440s.bsp`

Contains ephemeris data for planetary bodies. The version with "s" appended to the filename covers only more recent time (starts in the 1500s) to reduce filesize.

High Precision Earth Binary Planetary Constants Kernel (PCK)

e.g. `earth_000101_211220_210926.bpc`

Contains high precision orientation data for Earth in the ECEF ITRF93 reference frame.

ITRF93 is a more precise version of the standard IAU_EARTH reference frame provided in the text PCK below.

This kernel is regenerated several times per week so we should retrieve this from NAIF for every processing run.

ITFR93 Reference Frame Kernel for Earth

e.g. `earth_assoc_itrf93.tf`

Used to designate ITRF93 as the default body-fixed frame associated with the Earth.

Standard Text Planetary Constants Kernel (PCK)

e.g. `pck00010.tpc`

Contains orientation data and other planetary constants for planetary bodies.

2.10 Libera Data Models and Database Schema in DynamoDB on AWS

2.10.1 Under Review (Remove before release)

This document provides an overview of the data models used in the Libera databases in AWS DynamoDB. The Libera databases are designed to take advantage of the DynamoDB pricing model to minimize costs and utilize the serverless nature of DyanmoDB to facilitate an asynchronous event-driven architecture.

See *DynamoDB Basics in AWS* for more information on the basics of DynamoDB before diving into this page if you are unfamiliar with this AWS service.

This documentation will provide the examples of the data models for different DynamoDB tables used in the Libera SDC. Each visual table here will describe the key-value nature of the DynamoDB tables and how they are used in the Libera in the following way:

Partition Key	**Sort Key**	Attribute 1 Key	Attribute 2 Key	...	Attribute N Key
PK Value Example	SK Value Example	Value1 Example	Value2 Example	...	ValueN Ex.
PK Description	*SK Description*	*Attribute 1 Desc*	*Attribute 2 Desc*	...	*Attribute N Desc*

Note: Many tables in the Libera SDC use a generic key-value entry for the two keys (partition and sort) as follows:

- Partition Key - “PK”:”value”
- Sort Key- “SK”:”value”

This is in contrast to a more specifically named key-value entry that is more readable by a human, but can be restrictive in the DynamoDB table design as it relates to vertical partitioning and logical grouping of data. For example, below is a more human-readable key set that is not used in the Libera SDC, but is representative of logical example of how the keys are used.

- Partition Key - ‘Filename’:”Example_file.nc”
- Sort Key - ‘Filetype’:”L0”

This is generic usage of “PK” and “SK” is a common pattern in DynamoDB to allow for easy access to the data in the table and allows greater flexibility in the logical design of the data model without confusing semantics of the keys.

2.10.2 Libera Metadata and Provenance Database

2.10.3 Metadata Data Model

This database is used to store metadata and provenance information about the data files generated by the Libera SDC. The metadata for each file can broadly be broken down into three categories that are all stored together in the same table:

1. **File metadata** about the file itself, such as the filename, the time it was archived, and the version of the algorithm used to generate the data file
2. **Sortable metadata** about the file, such as the applicable date of the data in the file, calibration versions used, and other metadata that can be used to sort and filter the data in other processing steps
3. **Additional metadata** that is specific to a given data level. For example, the construction records for level 0 data files contain a wide array of additional metadata that is not present in higher level data products.

2.10.4 Minimum Database Examples

Additional attributes may be added as needed for specific data products, this shows the expected minimum.

File Metadata Example

PK	**SK**	archive-time	algorithm-version
“Example_file.nc”	“#”	“2024-01-01 00:00:00”	“1.0.0”
Unique Filename	*Placeholder*	*Archive Time*	*Algorithm Version*

Sortable Metadata Example

PK	SK	applicable-date
"Example_file.nc"	"#L0#SPICE#AZ"	"2024-01-01"
Unique Filename	*File Type Identifier*	*Applicable Date of Data*

Additional Metadata Example

No minimum requirement.

2.10.5 Level 0 (L0) Data Specifics

These are the raw binary data files received from ASDC and come as two separate files: construct records and PDS files.

Construction Record (CR)

CR File Metadata

PK	SK	ingest-time	archive-time	algorithm-version
"L0_CONS_file.PDS"	"#"	"2024-01-01 00:00:00"	"2024-01-01 00:01:00"	"1.0.0"
Unique Filename	*Placeholder*	*Applicable Date of Data*	*Archive Time*	*Algorithm Version*

CR Sortable Metadata

PK	SK	applicable-date	first-packet-time	last-packet-time	missing-packet-count	filled-gap-count
"L0_CONS_fil	"#L0#APID1	"2024-01-01"	"2024-01-01 00:00:00"	"2024-01-01 00:00:00"	0	0
Unique Filename	*File Type Identifier*	*Applicable Date of Data*	*First Packet Time*	*Last Packet Time*	*Missing Packet Count*	*Filled Gap Count*

CR Additional Metadata (Not Comprehensive of all Entries)

PK	SK	filename	edos-version	SCID	APID	...
"L0_CONS_file.PDS"	"#PDS"	"L0_PDS_file.PDS"				
Unique Filename	*Identifier*	*Applicable Date of Data*				
_____	_____	_____-	_____	___	_	_
"L0_CONS_file.PDS"	"#APID11"		"1.0.0"	1	11	...
Unique Filename	*Identifier*		*EDOS Version*	*SCID*	*APID*	...

The above example shows only 2 example items. The construction record file has approximately 10 more items associated with it that are stored in this table. See the `io/pds.py` file for the full details of items stored in the construction record.

PDS File

PDS File Metadata

This is the same as the construction record file metadata.

PK	SK	ingest-time	archive-time	algorithm-version
"L0_file.nc"	"#"	"2024-01-01 00:00:00"	"2024-01-01 00:01:00"	"1.0.0"
Unique Filename	*File Type*	*Applicable Date of Data*	*Archive Time*	*Algorithm Version*

PDS Sortable Metadata

This is covered in the construction record sortable metadata.

PDS Additional Metadata

This is covered by the construction record additional metadata.

2.10.6 Higher Level Data Products

TBD

2.10.7 Calibration Data Products

Cal File Metadata

PK	SK	archive-time	calibration-version
"Example_calibration_file.nc"	"#CAL#L0"	"2024-01-01 00:00:00"	"1.0.0"
Unique Filename	*Identifier*	*Archive Time*	*Calibration Version*

Cal Sortable Metadata

Not applicable.

Cal Additional Metadata

TBD.

2.10.8 Using this Data Model with a Global Secondary Index (GSI)

The Libera databases make use of a global secondary index (GSI) to allow for efficient querying of the data in the table using an alternate key. This is used to define specific "Use Cases" that allow for querying the data in different ways from the standard primary key of PK + SK.

NOTE: GSI's are not required to be unique with a primary key (PK + SK).

The GSI uses the existing attributes in the existing database and allows for a different way to access the data in the table in more efficient ways than scanning the whole database manually then sorting yourself.

Libera Metadata Database GSI's

1. Date searching GSI - This GSI is used to search for all files that have a specific date associated with them and get the filenames associated with particular identifier.

applicable-date	SK	PK
"2024-01-01"	"#L0"	"L0_CONS_file.PDS"
Applicable Date	*Type Identifier*	*Unique Filename*

2. Calibration Version searching GSI (TBD) - This GSI is used to search for all files that have a specific calibration identifier and retrieve the version associated with them.

SK ****	PK	version
"#Cal"	"Example_calibration_file.nc "	"1.0.0"
Type Identifier	*Unique Filename*	*Calibration Version*

3. Additional GSI's are TBD.

Implementation

Implementation specifics for general data products live in the `db/dynamodb_utils.py` and are under active development.

Specifics for level 0 (L0) raw data files are in the `io/pds.py` file. This is for construction records and PDS files received from ASDC, and these are used and refined in the `io/pds-ingest.py`.

The handling of this data model for higher level products is TBD with the basic structure and tools in place in the `db/dynamodb_utils.py` file.

2.11 DynamoDB Basics in AWS

This is a document that highlights key aspects of DynamoDB in AWS and how they are relevant to the Libera SDC. To understand the Libera data model, it is important to understand the basics of DynamoDB and how it works. This document provides an overview of the DynamoDB basics and how they are used in the Libera databases and is not intended to be a comprehensive guide to DynamoDB. For more information on DynamoDB, see the AWS DynamoDB documentation.

See the *Libera Data Models and Database Schema for DynamoDB in AWS* for more information on the specific data models used in the Libera databases.

2.11.1 DyanmoDB Formal Description

DynamoDB is a serverless NoSQL database service provided by AWS. It is a key-value and document database. It is a fully managed database with built-in security, backup and restore, and in-memory caching for internet-scale applications. DynamoDB uses tables, items, and attributes as the core components that you work with. A table is a collection of items, and each item is a collection of attributes. DynamoDB uses primary keys to uniquely identify each item in a table.

Important Note: The key-value nature of DynamoDB specifically uses only strings for keys and has a limited set of data types available for values. Libera SDC at this time only utilizes string and numeric data types in the DynamoDB tables.

Costs

At its simplest, DyanmoDB usage is billed based on the number and size of read and write operations. The design of the database schema can have a significant impact on the cost of using DynamoDB. The Libera databases are designed to take advantage of the DynamoDB pricing model to minimize costs with small read operations by using the concept of vertical scaling. See the vertical scaling section later in this documentation.

Event Driven Architecture

DynamoDB is designed for use in an event-driven architecture. This means that you can trigger events based on changes to the database. This is useful for triggering Lambda functions, SNS notifications, or other AWS services based on changes to the database. DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table and can pass these changes to a Lambda function for processing.

Libera specifically makes use of the DynamoDB Streams feature to trigger Lambda functions when changes are made to the database. This is used to coordinate downstream processing of different processing steps using Lambda and Step Functions. The data models for the Libera databases are designed to take advantage of this feature while stil using the vertical scaling pattern to minimize costs.

DynamoDB Primary Keys

To access specific items DynamoDB supports two types of primary keys to **uniquely** access items in a table:

- Partition key: A simple primary key, composed of one attribute known as the partition key.
- Partition key and sort key: A composite primary key, composed of two attributes. The first attribute is the partition key, and the second attribute is the sort key.

Libera uses the partition key and sort key access pattern.

Reading Data Well

DynamoDB allows for three different methods of reading data:

- GetItem: Retrieves a single item from a table.
- Query: Retrieves all items that have the same partition key.
- Scan: Retrieves all items in a table.

Note that the Query and Get operations are the most efficient way to access data in DynamoDB as it allows you to access data based on specific keys. The Scan operation is the least efficient way to access data as it reads all items in the table and can be very expensive. Avoid using the Scan operation whenever possible as the tables get large.

DynamoDB Secondary Indexes

DynamoDB supports secondary indexes to allow you to query the data in a table using an alternate key. This is used by Libera to define specific "Use Cases" that allow for querying the data in different ways from the standard primary key.

As a specific example, the Libera metadata database functions primarily as a record of each file that has been processed where the unique primary key in the database is the filename and one of the attributes is the date for which the data in the file was collected. If we wanted to access all files that relate to a specific date just using the primary key, we would need to scan the whole database then search on the specific date attribute. This would be very inefficient and potentially expensive. Instead, we can define a secondary index on the date attribute and then query the database using the date attribute as the key. This allows us to use a Query command to access the data we need and get back only the items in the database that have the specific date attribute of the date we are interested in. Further details on the specific secondary indexes used in the Libera databases are provided in the following sections. Further details on DynamoDB secondary indexes can be found in the AWS Secondary Index documentation.

Vertical Partitioning

Vertical partitioning is a database design pattern that involves effectively splitting a table into smaller tables based on the attributes that are most frequently accessed together. This can be used to optimize the performance of queries and reduce the cost of accessing data in the database. In the Libera databases, we use vertical partitioning with the sort key to allow for efficient access to the data in the database. This is done by defining the sort key as a string that is a concatenation of the attributes that are most frequently accessed together. This allows us to use the Query operation to access the data we need and get back only the items in the database that have the specific sort key attribute of the sort key we are interested in. This is a key design pattern used in the Libera databases to optimize the performance and cost of accessing data in the database. Further details on vertical partitioning can be found in the AWS Vertical Partitioning blog post.

CHAPTER

THREE

API

This is the API documentation for the entire Libera Utils library.

libera_utils	libera_utils

3.1 libera_utils

libera_utils

Modules

libera_utils.aws	
libera_utils.cli	Module for the Libera SDC utilities CLI
libera_utils.config	Configuration reader.
libera_utils.db	db module for dyanmodb operations.
libera_utils.geolocation	Module for performing geolocation tasks
libera_utils.io	
libera_utils.kernel_maker	Module containing CLI tool for creating SPICE kernels from packets
libera_utils.logutil	Logging utilities
libera_utils.packets	Module for reading packet data
libera_utils.quality_flags	Quality flag definitions
libera_utils.spice_utils	Modules for SPICE kernel creation, management, and usage
libera_utils.time	Module for dealing with time and time conventions
libera_utils.version	Module for anything related to package versioning

3.1.1 libera_utils.aws

Modules

libera_utils.aws.constants	AWS ECR Repository/Algorithm names
libera_utils.aws.ecr_upload	Module for uploading docker images to the ECR
libera_utils.aws.processing_step_function_tı	Module for manually triggering a step function
libera_utils.aws.utils	Helper functions for AWS access

libera_utils.aws.constants

AWS ECR Repository/Algorithm names

Classes

DataProductIdentifier(value[, names, ...])	Enumeration of data product canonical IDs used in AWS resource naming These IDs refer to the data products (files) themselves, NOT the processing steps (since processing steps may produce multiple products).
ProcessingStepIdentifier(value[, names, ...])	Enumeration of processing step IDs used in AWS resource naming and processing orchestration

libera_utils.aws.constants.DataProductIdentifier

class libera_utils.aws.constants.**DataProductIdentifier**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enumeration of data product canonical IDs used in AWS resource naming These IDs refer to the data products (files) themselves, NOT the processing steps (since processing steps may produce multiple products).

In general these names are of the form <level>-<source>-<type> # TODO: This enum is duplicated in libera_cdk in libera_lambda_runtime.constants

When that code is stable, it should be moved here and libera_lambda_runtime should import it from here.

__init__(**args*, ***kwds*)

Attributes

l0_rad_pds
l0_cam_pds
l0_az_pds
l0_el_pds
l0_jpss_pds
spice_az_ck
spice_el_ck
spice_jpss_ck
spice_jpss_spk
cal_rad
cal_cam
l1b_rad
l1b_cam
anc_adm

libera_utils.aws.constants.ProcessingStepIdentifier

class libera_utils.aws.constants.**ProcessingStepIdentifier**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enumeration of processing step IDs used in AWS resource naming and processing orchestration

In orchestration code, these are used as "NodeID" values to identify processing steps # TODO: this enum is duplicated in libera_cdk in the libera_lambda_runtime.constants module for ease of development

> When that is working well, independent of libera_utils, this enum should be replaced with that code and the Lambda runtime package should import it from here. 2024-04-30

__init__(**args*, ***kwds*)

Attributes

l2cf
l2_stf
adms
l2_surface_flux
l2_firf
unfilt
spice_az
spice_el
spice_jpss
pds_ingest
l1b_rad

class libera_utils.aws.constants.**DataProductIdentifier**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enumeration of data product canonical IDs used in AWS resource naming These IDs refer to the data products (files) themselves, NOT the processing steps (since processing steps may produce multiple products).

In general these names are of the form <level>-<source>-<type> # TODO: This enum is duplicated in libera_cdk in libera_lambda_runtime.constants

When that code is stable, it should be moved here and libera_lambda_runtime should import it from here.

class libera_utils.aws.constants.**ProcessingStepIdentifier**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enumeration of processing step IDs used in AWS resource naming and processing orchestration

In orchestration code, these are used as "NodeID" values to identify processing steps # TODO: this enum is duplicated in libera_cdk in the libera_lambda_runtime.constants module for ease of development

When that is working well, independent of libera_utils, this enum should be replaced with that code and the Lambda runtime package should import it from here. 2024-04-30

libera_utils.aws.ecr_upload

Module for uploading docker images to the ECR

Functions

login_to_ecr(account_id, region_name)	Login to the AWS ECR using commands :param account_id: Users AWS account ID :type account_id: int :param region_name: String of the region that the users AWS account is in :type region_name: string
upload_image_to_ecr(parsed_args)	Upload docker image to the correct ECR repository

libera_utils.aws.ecr_upload.login_to_ecr

libera_utils.aws.ecr_upload.**login_to_ecr**(*account_id*, *region_name*)

Login to the AWS ECR using commands :param account_id: Users AWS account ID :type account_id: int :param region_name: String of the region that the users AWS account is in :type region_name: string

Returns
result – subproccess object that holds the details of the completed CLI command

Return type
CompletedProcess

libera_utils.aws.ecr_upload.upload_image_to_ecr

libera_utils.aws.ecr_upload.**upload_image_to_ecr**(*parsed_args: Namespace*)

Upload docker image to the correct ECR repository

Parameters
parsed_args (*argparse.Namespace*) – Namespace of parsed CLI arguments

Return type
None

libera_utils.aws.ecr_upload.**login_to_ecr**(*account_id*, *region_name*)

Login to the AWS ECR using commands :param account_id: Users AWS account ID :type account_id: int :param region_name: String of the region that the users AWS account is in :type region_name: string

Returns
result – subproccess object that holds the details of the completed CLI command

Return type
CompletedProcess

libera_utils.aws.ecr_upload.**upload_image_to_ecr**(*parsed_args: Namespace*)

Upload docker image to the correct ECR repository

Parameters
parsed_args (*argparse.Namespace*) – Namespace of parsed CLI arguments

Return type
None

libera_utils.aws.processing_step_function_trigger

Module for manually triggering a step function

Functions

step_function_trigger(parsed_args)	Start a stepfunction to process a certain days data :param parsed_args: Namespace of parsed CLI arguments :type parsed_args: argparse.Namespace :param region_name: string of the AWS region name :type region_name: str

libera_utils.aws.processing_step_function_trigger.step_function_trigger

libera_utils.aws.processing_step_function_trigger.**step_function_trigger**(*parsed_args: Namespace*)

Start a stepfunction to process a certain days data :param parsed_args: Namespace of parsed CLI arguments :type parsed_args: argparse.Namespace :param region_name: string of the AWS region name :type region_name: str

Return type
None

libera_utils.aws.processing_step_function_trigger.**step_function_trigger**(*parsed_args: Namespace*)

Start a stepfunction to process a certain days data :param parsed_args: Namespace of parsed CLI arguments :type parsed_args: argparse.Namespace :param region_name: string of the AWS region name :type region_name: str

Return type
None

libera_utils.aws.utils

Helper functions for AWS access

Functions

get_aws_account_number([region_name])	Get a users AWS account ID number :param region_name: Region that the users AWS account is on :type region_name: string

libera_utils.aws.utils.get_aws_account_number

libera_utils.aws.utils.**get_aws_account_number**(*region_name='us-west-2'*)

Get a users AWS account ID number :param region_name: Region that the users AWS account is on :type region_name: string

Returns
account_id – users account_id number

Return type
int

libera_utils.aws.utils.**get_aws_account_number**(*region_name='us-west-2'*)

Get a users AWS account ID number :param region_name: Region that the users AWS account is on :type region_name: string

Returns
account_id – users account_id number

Return type
int

3.1.2 libera_utils.cli

Module for the Libera SDC utilities CLI

Functions

main([cli_args])	Main CLI entrypoint that runs the function inferred from the specified subcommand
parse_cli_args(cli_args)	Parse CLI arguments
print_version_info(*args)	Print CLI version information

libera_utils.cli.main

libera_utils.cli.**main**(*cli_args: list = None*)

Main CLI entrypoint that runs the function inferred from the specified subcommand

libera_utils.cli.parse_cli_args

libera_utils.cli.**parse_cli_args**(*cli_args: list*)

Parse CLI arguments

Parameters
cli_args (*list*) – List of CLI arguments to parse

Returns
Parsed arguments in a Namespace object

Return type
argparse.Namespace

libera_utils.cli.print_version_info

libera_utils.cli.**print_version_info**(**args*)

Print CLI version information

libera_utils.cli.**main**(*cli_args: list = None*)

Main CLI entrypoint that runs the function inferred from the specified subcommand

libera_utils.cli.**parse_cli_args**(*cli_args: list*)

Parse CLI arguments

Parameters
cli_args (*list*) – List of CLI arguments to parse

Returns
Parsed arguments in a Namespace object

Return type
argparse.Namespace

libera_utils.cli.**print_version_info**(**args*)

Print CLI version information

3.1.3 libera_utils.config

Configuration reader. To modify the configuration, see file: config.json

Module Attributes

config	Singleton (one per process) accessor for *libera_utils.config._ConfigurationCache()*

libera_utils.config.config

libera_utils.config.**config = <libera_utils.config._ConfigurationCache object>**

Singleton (one per process) accessor for *libera_utils.config._ConfigurationCache()*

Classes

ConfigurationFormatter()	Customize the string formatter to replace fields in a config string with values from the configuration dictionary.

libera_utils.config.ConfigurationFormatter

class libera_utils.config.**ConfigurationFormatter**

Bases: Formatter

Customize the string formatter to replace fields in a config string with values from the configuration dictionary. This will allow configuration parameters in the emus_config.json file to be based off of other configuration parameters by wrapping the configuration key in curly braces.

Methods

get_field(field_name, args, kwargs)	
get_value(key, *args, **kwargs)	Overrides the default get_value method in the python formatter.
parse(format_string)	

check_unused_args
convert_field
format
format_field
vformat

__init__(**args*, ***kwargs*)

Methods

get_value(key, *args, **kwargs)	Overrides the default get_value method in the python formatter.

get_value(*key: str*, **args*, ***kwargs*)

Overrides the default get_value method in the python formatter. This will return the value from the emus configuration with the specified key.

class libera_utils.config.**ConfigurationFormatter**

Customize the string formatter to replace fields in a config string with values from the configuration dictionary. This will allow configuration parameters in the emus_config.json file to be based off of other configuration parameters by wrapping the configuration key in curly braces.

Methods

get_field(field_name, args, kwargs)	
get_value(key, *args, **kwargs)	Overrides the default get_value method in the python formatter.
parse(format_string)	

check_unused_args
convert_field
format
format_field
vformat

get_value(*key: str*, **args*, ***kwargs*)

Overrides the default get_value method in the python formatter. This will return the value from the emus configuration with the specified key.

class libera_utils.config.**_ConfigurationCache**

Class that stores the JSON configuration and provides methods for accessing configuration information

Methods

force_reload()	Force reloading of the JSON config
get(key)	Retrieves a configuration value from either the cached JSON or from the environment

_format_return_value(*value: str*)

Recursively formats the returned value, looking for config keys to substitute.

Parameters
value (*str*) – String to format

Return type
str

_parse_numeric_types(*value: str*)

Checks the final result of a config retrieval. If it is a string that can be interpreted as a float or int, parse it and return that.

Parameters
value (*any*) – Final formatted value.

Return type
str or float or int

force_reload()

Force reloading of the JSON config

get(*key*)

Retrieves a configuration value from either the cached JSON or from the environment

Parameters
key (*str*) – Key for which to retrieve the configured value.

Returns
Resulting value

Return type
any

libera_utils.config.**config = <libera_utils.config._ConfigurationCache object>**

Singleton (one per process) accessor for *libera_utils.config._ConfigurationCache()*

3.1.4 libera_utils.db

db module for dyanmodb operations.

Modules

libera_utils.db.dynamodb_utils	Module for database utilities

libera_utils.db.dynamodb_utils

Module for database utilities

Functions

add_archive_time_to_ddb_item(ddb_item)	Add archive time to DynamoDB item
create_ddb_metadata_applicable_date_item(...)	Write metadata record to DynamoDB for a single file
create_ddb_metadata_file_item(filename, ...)	Write metadata record to DynamoDB for a single file
get_dynamodb_table(dynamo_table_name)	Get the DynamoDB table

libera_utils.db.dynamodb_utils.add_archive_time_to_ddb_item

libera_utils.db.dynamodb_utils.**add_archive_time_to_ddb_item**(*ddb_item: dict*)

Add archive time to DynamoDB item

libera_utils.db.dynamodb_utils.create_ddb_metadata_applicable_date_item

libera_utils.db.dynamodb_utils.**create_ddb_metadata_applicable_date_item**(*filename: str*, *data_level: str*, *data_type: str*, *applicable_date: str*, *data_subtype: str = None*, *additional_metadata: dict = None*)

Write metadata record to DynamoDB for a single file

libera_utils.db.dynamodb_utils.create_ddb_metadata_file_item

libera_utils.db.dynamodb_utils.**create_ddb_metadata_file_item**(*filename: str*, *algorithm_version: str*, *include_archive_time: bool = False*, *additional_metadata: dict = None*)

Write metadata record to DynamoDB for a single file

libera_utils.db.dynamodb_utils.get_dynamodb_table

libera_utils.db.dynamodb_utils.**get_dynamodb_table**(*dynamo_table_name: str*)

Get the DynamoDB table

libera_utils.db.dynamodb_utils.**add_archive_time_to_ddb_item**(*ddb_item: dict*)

Add archive time to DynamoDB item

libera_utils.db.dynamodb_utils.**create_ddb_metadata_applicable_date_item**(*filename: str*, *data_level: str*, *data_type: str*, *applicable_date: str*, *data_subtype: str = None*, *additional_metadata: dict = None*)

Write metadata record to DynamoDB for a single file

libera_utils.db.dynamodb_utils.**create_ddb_metadata_file_item**(*filename: str*, *algorithm_version: str*, *include_archive_time: bool = False*, *additional_metadata: dict = None*)

Write metadata record to DynamoDB for a single file

libera_utils.db.dynamodb_utils.**get_dynamodb_table**(*dynamo_table_name: str*)

Get the DynamoDB table

3.1.5 libera_utils.geolocation

Module for performing geolocation tasks

Functions

angle_between(v1, v2[, degrees])	Returns angle between vectors v1 and v2, in units of radians (default) or degrees.
cartesian_to_planetographic(cartesian_coords)	Convert cartesian coordinates in the ITRF93 frame to planetographic latitude and longitude.
frame_transform(from_frame, to_frame, et, ...)	Transform a position <x, y, z> vector between reference frames, optionally normalizing the result.
get_earth_radii()	Retrieve Earth radii values from SPICE
sub_observer_point(target, et, frame, observer)	Computes the cartesian coordinates of the sub-observer point at time *et* and the observer altitude above the point.
sub_solar_point(target, et, frame, observer)	Computes the cartesian coordinates of the subsolar point at ephemeris time et.
surface_intercept_point(sc_location, ...[, et])	Returns rectangular coordinates of the point of interception of a look direction from the spacecraft onto the Earth ellipsoid.
target_position(target, et, frame, observer)	Calculates the position and velocity of the *target* at ephemeris time *et* relative to *observer* in reference frame *frame*.

libera_utils.geolocation.angle_between

libera_utils.geolocation.**angle_between**(*v1: ndarray*, *v2: ndarray*, *degrees: bool = False*)

Returns angle between vectors v1 and v2, in units of radians (default) or degrees. N is the number of vectors D is the dimension of the space

Parameters

- **v1** (*numpy.ndarray*) – Vector(s) 1. May be shape (D,) or (N, D).
- **v2** (*numpy.ndarray*) – Vector(s) 2. May be shape (D,) or (N, D).
- **degrees** (*bool*) – Specify True to return result in degrees. Default is False (returns radians).

Returns

Angle between v1 and v2 in radians (optionally in degrees)

Return type

float or numpy.ndarray

libera_utils.geolocation.cartesian_to_planetographic

libera_utils.geolocation.**cartesian_to_planetographic**(*cartesian_coords: ndarray*, *degrees: bool = True*)

Convert cartesian coordinates in the ITRF93 frame to planetographic latitude and longitude. Longitude runs 0-360 such that longitude appears to increase as the planet rotates when viewed by an observer and latitude is calculated from a surface normal vector rather than a line through the planet center. See https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/17_frames_and_coordinate_systems.pdf for reference.

Parameters

- **cartesian_coords** (*numpy.ndarray*) – Rectangular coordinates in ITRF93 frame.
- **degrees** (*bool*) – Default true. If False, returns angles in radians.

Returns
Each coordinate is returned as (longitude, latitude, altitude).

Return type
numpy.ndarray

libera_utils.geolocation.frame_transform

libera_utils.geolocation.**frame_transform**(*from_frame: SpiceFrame*, *to_frame: SpiceFrame*, *et: float*, *position: ndarray*, *normalize: bool = False*) → ndarray

Transform a position <x, y, z> vector between reference frames, optionally normalizing the result.

Parameters

- **from_frame** (*spice_utils.SpiceFrame*) – Reference frame of position
- **to_frame** (*spice_utils.SpiceFrame*) – Reference frame of output
- **et** (*numpy.float64 or numpy.ndarray*) – Ephemeris time(s) corresponding to position(s). For time-independent transformations, this can by any valid ephemeris time.
- **position** (*numpy.ndarray*) – <x, y, z> vector or array of vectors in reference frame *from_frame*
- **normalize** (*bool, Optional*) – Optionally normalize the output vector

Returns
3d position vector(s) in reference frame *to_frame*

Return type
numpy.ndarray

libera_utils.geolocation.get_earth_radii

libera_utils.geolocation.**get_earth_radii**()

Retrieve Earth radii values from SPICE

Returns
(re, rp, flat) tuple of equatorial and polar ellipsoid radii and a flattening coefficient

Return type
tuple

libera_utils.geolocation.sub_observer_point

libera_utils.geolocation.**sub_observer_point**(*target:* SpiceBody, *et:* *float*, *frame:* SpiceFrame, *observer:* SpiceBody, *abcorr:* *str* = *'NONE'*, *method:* *str* = *'NEAR POINT/ELLIPSOID'*)

Computes the cartesian coordinates of the sub-observer point at time *et* and the observer altitude above the point. Units in km.

Parameters

- **target** (spice_utils.SpiceBody) – Body on which the sub point will be calculated (usually a planetary body).
- **et** (*float or numpy.ndarray*) – Ephemeris time of observation.
- **frame** (spice_utils.SpiceFrame) – Reference frame for returned vectors.
- **observer** (spice_utils.SpiceBody) – The object from which to calculate the sub point (e.g. a spacecraft). A-B correction is applied based on the distance between observer and sub observer point.
- **abcorr** (*str, Optional*) – String delineating what kind of A-B light time correction to perform. Default is 'NONE'.
- **method** (*str, Optional*) – String specifying what kind of method to use to find the vector between the observer and the target. Default is *NEAR POINT/ELLIPSOID*, which uses the nearest point on the ellipsoid rather than drawing a line through the center of the ellipsoid.

Returns

Cartesian point on the target body surface in the specified reference frame and also the euclidean distance between the observer and the sub point in order: [x, y, z], obs_alt

Return type

numpy.ndarray, float

libera_utils.geolocation.sub_solar_point

libera_utils.geolocation.**sub_solar_point**(*target:* SpiceBody, *et:* *float*, *frame:* SpiceFrame, *observer:* SpiceBody, *abcorr='LT+S'*, *method='NEAR POINT/ELLIPSOID'*)

Computes the cartesian coordinates of the subsolar point at ephemeris time et.

Parameters

- **target** (spice_utils.SpiceBody) – Body on which the sub point will be calculated (usually a planetary body).
- **et** (*float or numpy.ndarray*) – Ephemeris time of observation.
- **frame** (spice_utils.SpiceFrame) – Reference frame for returned vectors.
- **observer** (spice_utils.SpiceBody) – The object from which to calculate the subsolar point (e.g. a spacecraft). A-B correction is applied based on the distance between observer and subsolar point.
- **abcorr** (*str*) – String delineating what kind of A-B lighttime correction to perform. Default is 'LT+S'.
- **method** (*str*) – String specifying what kind of method to use to find the vector between the observer and the target. Default is *NEAR POINT/ELLIPSOID*, which uses the nearest point on the ellipsoid rather than drawing a line through the center of the ellipsoid.

Returns
Subsolar point on the ellipsoid surface in the specified reference frame, apparent epoch at that point (depending on specified light time correction), and vector from observer to subsolar point.

Return type
numpy.ndarray, numpy.ndarray, numpy.ndarray

libera_utils.geolocation.surface_intercept_point

libera_utils.geolocation.**surface_intercept_point**(*sc_location: ndarray*, *look_vector: ndarray*, *look_frame:* SpiceFrame, *et: float = None*)

Returns rectangular coordinates of the point of interception of a look direction from the spacecraft onto the Earth ellipsoid. If the look vector misses the planet, then the distance returned will be non-zero and the point returned is the point on the look_vector ray that is closest to the ellipsoid.

This routine assumes that the location of the spacecraft and the location of the instrument are the same because we don't have ephemeris data for the instrument but we _do_ have ephemeris for the spacecraft. Over the scale of distances involved, the offset between spacecraft and instrument (meters) should be negligible in affecting the near-point calculation.

https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/npedln_c.html

Parameters

- **sc_location** (*numpy.ndarray*) – The location of the observing body (i.e. the spacecraft body) with respect to Earth
- **look_vector** (*numpy.ndarray*) – Look direction unit vector (e.g. an instrument look direction)
- **look_frame** (spice_utils.SpiceFrame) – Reference frame of *look_vector*
- **et** (*float or numpy.ndarray or None, Optional*) – Ephemeris time (at spacecraft at photon detection time). Only required if look_frame is not ITRF93.

Returns
(pnear, alt) Rectangular coordinates of nearest point to reference surface ellipsoid and distance between the line and the near point.

Return type
tuple

libera_utils.geolocation.target_position

libera_utils.geolocation.**target_position**(*target:* SpiceBody, *et: float*, *frame:* SpiceFrame, *observer:* SpiceBody, *abcorr: str = 'NONE'*, *normalize: bool = False*)

Calculates the position and velocity of the *target* at ephemeris time *et* relative to *observer* in reference frame *frame*. Also calculates the light travel time between *target* and *observer* at time *et*.

Parameters

- **target** (spice_utils.SpiceBody) – Target body for which to calculate position and velocity relative to observer
- **et** (*float or numpy.ndarray*) – Ephemeris time(s)
- **frame** (spice_utils.SpiceFrame) – Reference frame (unit vectors)

- **observer** (spice_utils.SpiceBody) – The observer of the target. Resulting coordinates point from observer to target.
- **abcorr** (*str*) – A scalar string that indicates the aberration corrections to apply to the database of the target body to account for one-way light time and stellar aberration. Default is 'NONE'.
- **normalize** (*bool, Optional*) – Return unit vectors for position and velocity (light time output is unchanged)

Returns

(x: numpy.ndarray, v: numpy.ndarray, lt: numpy.ndarray) or (x: float, v: float, lt: float) Rectangular position and velocity vectors (x, y, z), (v_x, v_y, v_z) where position points from the planet center of mass location at `et` to the aberration-corrected location of the target. Light time (lt) between planetary body and target.

Return type

tuple

libera_utils.geolocation.**angle_between**(*v1: ndarray*, *v2: ndarray*, *degrees: bool = False*)

Returns angle between vectors v1 and v2, in units of radians (default) or degrees. N is the number of vectors D is the dimension of the space

Parameters

- **v1** (*numpy.ndarray*) – Vector(s) 1. May be shape (D,) or (N, D).
- **v2** (*numpy.ndarray*) – Vector(s) 2. May be shape (D,) or (N, D).
- **degrees** (*bool*) – Specify True to return result in degrees. Default is False (returns radians).

Returns

Angle between v1 and v2 in radians (optionally in degrees)

Return type

float or numpy.ndarray

libera_utils.geolocation.**cartesian_to_planetographic**(*cartesian_coords: ndarray*, *degrees: bool = True*)

Convert cartesian coordinates in the ITRF93 frame to planetographic latitude and longitude. Longitude runs 0-360 such that longitude appears to increase as the planet rotates when viewed by an observer and latitude is calculated from a surface normal vector rather than a line through the planet center. See https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/17_frames_and_coordinate_systems.pdf for reference.

Parameters

- **cartesian_coords** (*numpy.ndarray*) – Rectangular coordinates in ITRF93 frame.
- **degrees** (*bool*) – Default true. If False, returns angles in radians.

Returns

Each coordinate is returned as (longitude, latitude, altitude).

Return type

numpy.ndarray

libera_utils.geolocation.**frame_transform**(*from_frame: SpiceFrame*, *to_frame: SpiceFrame*, *et: float*, *position: ndarray*, *normalize: bool = False*) → ndarray

Transform a position <x, y, z> vector between reference frames, optionally normalizing the result.

Parameters

- **from_frame** (spice_utils.SpiceFrame) – Reference frame of position

- **to_frame** (spice_utils.SpiceFrame) – Reference frame of output
- **et** (*numpy.float64 or numpy.ndarray*) – Ephemeris time(s) corresponding to position(s). For time-independent transformations, this can by any valid ephemeris time.
- **position** (*numpy.ndarray*) – <x, y, z> vector or array of vectors in reference frame *from_frame*
- **normalize** (*bool, Optional*) – Optionally normalize the output vector

Returns
3d position vector(s) in reference frame *to_frame*

Return type
numpy.ndarray

libera_utils.geolocation.**get_earth_radii**()

Retrieve Earth radii values from SPICE

Returns
(re, rp, flat) tuple of equatorial and polar ellipsoid radii and a flattening coefficient

Return type
tuple

libera_utils.geolocation.**sub_observer_point**(*target:* SpiceBody, *et: float*, *frame:* SpiceFrame, *observer:* SpiceBody, *abcorr: str = 'NONE'*, *method: str = 'NEAR POINT/ELLIPSOID'*)

Computes the cartesian coordinates of the sub-observer point at time *et* and the observer altitude above the point. Units in km.

Parameters

- **target** (spice_utils.SpiceBody) – Body on which the sub point will be calculated (usually a planetary body).
- **et** (*float or numpy.ndarray*) – Ephemeris time of observation.
- **frame** (spice_utils.SpiceFrame) – Reference frame for returned vectors.
- **observer** (spice_utils.SpiceBody) – The object from which to calculate the sub point (e.g. a spacecraft). A-B correction is applied based on the distance between observer and sub observer point.
- **abcorr** (*str, Optional*) – String delineating what kind of A-B light time correction to perform. Default is 'NONE'.
- **method** (*str, Optional*) – String specifying what kind of method to use to find the vector between the observer and the target. Default is *NEAR POINT/ELLIPSOID*, which uses the nearest point on the ellipsoid rather than drawing a line through the center of the ellipsoid.

Returns
Cartesian point on the target body surface in the specified reference frame and also the euclidean distance between the observer and the sub point in order: [x, y, z], obs_alt

Return type
numpy.ndarray, float

libera_utils.geolocation.**sub_solar_point**(*target:* SpiceBody, *et: float*, *frame:* SpiceFrame, *observer:* SpiceBody, *abcorr='LT+S'*, *method='NEAR POINT/ELLIPSOID'*)

Computes the cartesian coordinates of the subsolar point at ephemeris time et.

Parameters

- **target** (spice_utils.SpiceBody) – Body on which the sub point will be calculated (usually a planetary body).
- **et** (*float or numpy.ndarray*) – Ephemeris time of observation.
- **frame** (spice_utils.SpiceFrame) – Reference frame for returned vectors.
- **observer** (spice_utils.SpiceBody) – The object from which to calculate the subsolar point (e.g. a spacecraft). A-B correction is applied based on the distance between observer and subsolar point.
- **abcorr** (*str*) – String delineating what kind of A-B lighttime correction to perform. Default is 'LT+S'.
- **method** (*str*) – String specifying what kind of method to use to find the vector between the observer and the target. Default is *NEAR POINT/ELLIPSOID*, which uses the nearest point on the ellipsoid rather than drawing a line through the center of the ellipsoid.

Returns

Subsolar point on the ellipsoid surface in the specified reference frame, apparent epoch at that point (depending on specified light time correction), and vector from observer to subsolar point.

Return type

numpy.ndarray, numpy.ndarray, numpy.ndarray

libera_utils.geolocation.**surface_intercept_point**(*sc_location: ndarray*, *look_vector: ndarray*, *look_frame:* SpiceFrame, *et: float = None*)

Returns rectangular coordinates of the point of interception of a look direction from the spacecraft onto the Earth ellipsoid. If the look vector misses the planet, then the distance returned will be non-zero and the point returned is the point on the look_vector ray that is closest to the ellipsoid.

This routine assumes that the location of the spacecraft and the location of the instrument are the same because we don't have ephemeris data for the instrument but we _do_ have ephemeris for the spacecraft. Over the scale of distances involved, the offset between spacecraft and instrument (meters) should be negligible in affecting the near-point calculation.

https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/npedln_c.html

Parameters

- **sc_location** (*numpy.ndarray*) – The location of the observing body (i.e. the spacecraft body) with respect to Earth
- **look_vector** (*numpy.ndarray*) – Look direction unit vector (e.g. an instrument look direction)
- **look_frame** (spice_utils.SpiceFrame) – Reference frame of *look_vector*
- **et** (*float or numpy.ndarray or None, Optional*) – Ephemeris time (at spacecraft at photon detection time). Only required if look_frame is not ITRF93.

Returns

(pnear, alt) Rectangular coordinates of nearest point to reference surface ellipsoid and distance between the line and the near point.

Return type

tuple

libera_utils.geolocation.**target_position**(*target:* SpiceBody, *et: float*, *frame:* SpiceFrame, *observer:* SpiceBody, *abcorr: str = 'NONE'*, *normalize: bool = False*)

Calculates the position and velocity of the *target* at ephemeris time *et* relative to *observer* in reference frame *frame*. Also calculates the light travel time between *target* and *observer* at time *et*.

Parameters

- **target** (spice_utils.SpiceBody) – Target body for which to calculate position and velocity relative to observer
- **et** (*float or numpy.ndarray*) – Ephemeris time(s)
- **frame** (spice_utils.SpiceFrame) – Reference frame (unit vectors)
- **observer** (spice_utils.SpiceBody) – The observer of the target. Resulting coordinates point from observer to target.
- **abcorr** (*str*) – A scalar string that indicates the aberration corrections to apply to the database of the target body to account for one-way light time and stellar aberration. Default is 'NONE'.
- **normalize** (*bool, Optional*) – Return unit vectors for position and velocity (light time output is unchanged)

Returns

(x: numpy.ndarray, v: numpy.ndarray, lt: numpy.ndarray) or (x: float, v: float, lt: float) Rectangular position and velocity vectors (x, y, z), (v_x, v_y, v_z) where position points from the planet center of mass location at `et` to the aberration-corrected location of the target. Light time (lt) between planetary body and target.

Return type

tuple

3.1.6 libera_utils.io

Modules

libera_utils.io.caching	Module containing code to manage local file caching
libera_utils.io.filenaming	Module for file naming utilities
libera_utils.io.hdf	Utils for HDF5 file handling
libera_utils.io.manifest	Module for manifest file handling
libera_utils.io.smart_open	Module for smart_open

libera_utils.io.caching

Module containing code to manage local file caching

Functions

empty_local_cache_dir()	Remove all cached files in the local cache.
get_local_cache_dir()	Determine where to cache files based on the system and installed package version.

libera_utils.io.caching.empty_local_cache_dir

libera_utils.io.caching.**empty_local_cache_dir**()

Remove all cached files in the local cache.

Returns
List of removed files

Return type
list

libera_utils.io.caching.get_local_cache_dir

libera_utils.io.caching.**get_local_cache_dir**()

Determine where to cache files based on the system and installed package version.

Returns
Path to the cache directory for this version of this package on the current system

Return type
pathlib.Path

libera_utils.io.caching.**empty_local_cache_dir**()

Remove all cached files in the local cache.

Returns
List of removed files

Return type
list

libera_utils.io.caching.**get_local_cache_dir**()

Determine where to cache files based on the system and installed package version.

Returns
Path to the cache directory for this version of this package on the current system

Return type
pathlib.Path

libera_utils.io.filenaming

Module for file naming utilities

Functions

format_semantic_version(semantic_version)	Formats a semantic version string X.Y.Z into a filename-compatible string like VX-Y-Z, for X = major version, Y = minor version, Z = patch.
get_current_revision_str()	Get the current *r%y%j%H%M%S* string for filename revisions.
get_current_version_str(package_name)	Retrieve the current version of a (algorithm) package and format it for inclusion in a filename

libera_utils.io.filenaming.format_semantic_version

libera_utils.io.filenaming.**format_semantic_version**(*semantic_version: str*)

Formats a semantic version string X.Y.Z into a filename-compatible string like VX-Y-Z, for X = major version, Y = minor version, Z = patch.

Result is uppercase. Release candidate suffixes are allowed as no strict checking is done on the contents of X, Y, or Z. e.g. 1.2.3rc1 becomes V1-2-3RC1

Parameters
semantic_version (*str*) – String matching X.Y.Z where X, Y and Z are integers of any length

Return type
str

libera_utils.io.filenaming.get_current_revision_str

libera_utils.io.filenaming.**get_current_revision_str**()

Get the current *r%y%j%H%M%S* string for filename revisions.

Returns
Current (now) revision string.

Return type
str

libera_utils.io.filenaming.get_current_version_str

libera_utils.io.filenaming.**get_current_version_str**(*package_name: str*)

Retrieve the current version of a (algorithm) package and format it for inclusion in a filename

Parameters
package_name (*str*) – Package for which to retrieve a version string. This should be your algorithm package and it must use a semantic versioning scheme, configured in project metadata.

Returns
Version string in format vM1m2p3

Return type
str

Classes

AbstractValidFilename(*args, **kwargs)	Composition of a CloudPath/Path instance with some methods to perform regex validation on filenames
AnyFilename(*args, **kwargs)	Polymorphic class for creating a Filename object
AttitudeKernelFilename(*args, **kwargs)	Class to construct, store, and manipulate an SPK filename
DataLevel(value[, names, module, qualname, ...])	Data product level
EphemerisKernelFilename(*args, **kwargs)	Class to construct, store, and manipulate an SPK filename
L0Filename(*args, **kwargs)	Filename validation class for L0 files from EDOS.
LiberaDataProductFilename(*args, **kwargs)	Filename validation class for L1B and L2 science products
ManifestFilename(*args, **kwargs)	Class for naming manifest files
ManifestType(value[, names, module, ...])	Enumerated legal manifest type values

libera_utils.io.filenaming.AbstractValidFilename

class libera_utils.io.filenaming.**AbstractValidFilename**(**args*, ***kwargs*)

Bases: ABC

Composition of a CloudPath/Path instance with some methods to perform regex validation on filenames

Attributes

archive_prefix
Property that contains the generated prefix used for archiving, when applicable

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(*args[, basepath])	Abstract method that must be implemented to provide hinting for required parts
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

__init__(**args*, ***kwargs*)

Methods

from_filename_parts(*args[, basepath])	Abstract method that must be implemented to provide hinting for required parts
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

Attributes

archive_prefix	Property that contains the generated prefix used for archiving, when applicable
filename_parts	Property that contains a namespace of filename parts
path	Property containing the file path

static _calculate_applicable_time(*start: datetime*, *end: datetime*)

Based on the start time and end time of a file, returns the applicable time (date)

Parameters

- **start** (*datetime.datetime*) – Start of the applicable time range
- **end** (*datetime.datetime*) – End of the applicable time range

Returns
The date of the mean time between start and end

Return type
datetime.date

abstract classmethod _format_filename_parts(***parts*)

Format parts into a filename

Note: When this is implemented by concrete classes, **parts becomes a set of explicitly named arguments

classmethod _from_filename_parts(*, *basepath: str | Path | S3Path = None*, ***parts: Any*)

Create instance from filename parts.

The part kwarg names are named according to the regex for the file type.

Parameters

- **basepath** (*Union[str, Path, S3Path], Optional*) – Allows prepending a basepath or prefix.
- **parts** (*Any*) – Passed directly to _format_filename_parts. This is a dict of variable kwargs that will differ in each filename class based on the required parts for that particular filename type.

Return type
AbstractValidFilename

abstract _parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

abstract property archive_prefix

Property that contains the generated prefix used for archiving, when applicable

property filename_parts

Property that contains a namespace of filename parts

abstract classmethod from_filename_parts(**args: Any*, *basepath: str | Path | S3Path = None*, ***kwargs: Any*)

Abstract method that must be implemented to provide hinting for required parts

generate_prefixed_path(*parent_path: str | Path | S3Path*) → Path | S3Path

Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)

Parameters
parent_path (*Union[str, Path, S3Path]*) – Absolute path to the parent directory or S3 bucket prefix. The generated path prefix is appended to the parent path and followed by the file basename.

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

property path: Path | S3Path

Property containing the file path

regex_match(*path: str | Path | S3Path*)

Parse and validate a given path against class-attribute defined regex

Returns
Match group dict of filename parts

Return type
dict

libera_utils.io.filenaming.AnyFilename

class libera_utils.io.filenaming.**AnyFilename**(**args*, ***kwargs*)

Bases: object

Polymorphic class for creating a Filename object

__init__(**args*, ***kwargs*)

libera_utils.io.filenaming.AttitudeKernelFilename

class libera_utils.io.filenaming.**AttitudeKernelFilename**(**args*, ***kwargs*)

Bases: *AbstractValidFilename*

Class to construct, store, and manipulate an SPK filename

Attributes

archive_prefix
Property that contains the generated prefix for SPICE archiving

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(ck_object, version, ...)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

__init__(**args*, ***kwargs*)

Methods

from_filename_parts(ck_object, version, ...)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

Attributes

archive_prefix	Property that contains the generated prefix for SPICE archiving
filename_parts	Property that contains a namespace of filename parts
path	Property containing the file path

static _calculate_applicable_time(*start: datetime*, *end: datetime*)

Based on the start time and end time of a file, returns the applicable time (date)

Parameters

- **start** (*datetime.datetime*) – Start of the applicable time range
- **end** (*datetime.datetime*) – End of the applicable time range

Returns
The date of the mean time between start and end

Return type
datetime.date

classmethod _format_filename_parts(*ck_object: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*)

Format filename parts as a string

Parameters

- **ck_object** (*str*) – Name of object whose attitude is represented in this CK.
- **utc_start** (*datetime.datetime*) – Start time of data.
- **utc_end** (*datetime.datetime*) – End time of data.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **revision** (*datetime.datetime*) – When the file was last revised.

Return type
str

classmethod _from_filename_parts(*, *basepath: str | Path | S3Path = None*, ***parts: Any*)

Create instance from filename parts.

The part kwarg names are named according to the regex for the file type.

Parameters

- **basepath** (*Union[str, Path, S3Path], Optional*) – Allows prepending a basepath or prefix.
- **parts** (*Any*) – Passed directly to _format_filename_parts. This is a dict of variable kwargs that will differ in each filename class based on the required parts for that particular filename type.

Return type
AbstractValidFilename

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Property that contains the generated prefix for SPICE archiving

property filename_parts

Property that contains a namespace of filename parts

classmethod from_filename_parts(*ck_object: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*, *basepath: str | Path | S3Path | None = None*)

Create instance from filename parts.

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part arg names are named according to the regex for the file type.

Parameters

- **ck_object** (*str*) – Name of object whose attitude is represented in this CK.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – Start time of data.
- **utc_end** (*datetime.datetime*) – End time of data.
- **revision** (*datetime.datetime*) – When the file was last revised.
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
AttitudeKernelFilename

generate_prefixed_path(*parent_path: str | Path | S3Path*) → Path | S3Path

Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)

Parameters
parent_path (*Union[str, Path, S3Path]*) – Absolute path to the parent directory or S3 bucket prefix. The generated path prefix is appended to the parent path and followed by the file basename.

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

property path: Path | S3Path

Property containing the file path

regex_match(*path: str | Path | S3Path*)

Parse and validate a given path against class-attribute defined regex

Returns
Match group dict of filename parts

Return type
dict

libera_utils.io.filenaming.DataLevel

class libera_utils.io.filenaming.**DataLevel**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Data product level

__init__(**args*, ***kwds*)

Attributes

L0
SPICE
CAL
L1B
L2

libera_utils.io.filenaming.EphemerisKernelFilename

class libera_utils.io.filenaming.**EphemerisKernelFilename**(**args*, ***kwargs*)

Bases: *AbstractValidFilename*

Class to construct, store, and manipulate an SPK filename

Attributes

archive_prefix
Property that contains the generated prefix for SPICE archiving

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(spk_object, version, ...)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

__init__(**args*, ***kwargs*)

Methods

from_filename_parts(spk_object, version, ...)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

Attributes

archive_prefix	Property that contains the generated prefix for SPICE archiving
filename_parts	Property that contains a namespace of filename parts
path	Property containing the file path

static _calculate_applicable_time(*start: datetime*, *end: datetime*)

Based on the start time and end time of a file, returns the applicable time (date)

Parameters

- **start** (*datetime.datetime*) – Start of the applicable time range
- **end** (*datetime.datetime*) – End of the applicable time range

Returns
The date of the mean time between start and end

Return type
datetime.date

classmethod _format_filename_parts(*spk_object: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*)

Format filename parts as a string

Parameters

- **spk_object** (*str*) – Name of object whose ephemeris is represented in this SPK.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – Start time of data.
- **utc_end** (*datetime.datetime*) – End time of data.
- **revision** (*datetime.datetime*) – Time when the file was last revised

Return type
str

classmethod _from_filename_parts(*, *basepath: str | Path | S3Path = None*, ***parts: Any*)

Create instance from filename parts.

The part kwarg names are named according to the regex for the file type.

Parameters

- **basepath** (*Union[str, Path, S3Path], Optional*) – Allows prepending a basepath or prefix.
- **parts** (*Any*) – Passed directly to _format_filename_parts. This is a dict of variable kwargs that will differ in each filename class based on the required parts for that particular filename type.

Return type
AbstractValidFilename

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Property that contains the generated prefix for SPICE archiving

property filename_parts

Property that contains a namespace of filename parts

classmethod from_filename_parts(*spk_object: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*, *basepath: str | Path | S3Path | None = None*)

Create instance from filename parts.

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part arg names are named according to the regex for the file type.

Parameters

- **spk_object** (*str*) – Name of object whose attitude is represented in this SPK.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – Start time of data.
- **utc_end** (*datetime.datetime*) – End time of data.
- **revision** (*datetime.datetime*) – When the file was last revised.
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
EphemerisKernelFilename

generate_prefixed_path(*parent_path: str | Path | S3Path*) → Path | S3Path

Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)

Parameters
parent_path (*Union[str, Path, S3Path]*) – Absolute path to the parent directory or S3 bucket prefix. The generated path prefix is appended to the parent path and followed by the file basename.

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

property path: Path | S3Path

Property containing the file path

regex_match(*path: str | Path | S3Path*)

Parse and validate a given path against class-attribute defined regex

Returns
Match group dict of filename parts

Return type
dict

libera_utils.io.filenaming.L0Filename

class libera_utils.io.filenaming.**L0Filename**(**args*, ***kwargs*)

Bases: *AbstractValidFilename*

Filename validation class for L0 files from EDOS.

Attributes

archive_prefix
Property that contains the generated prefix for L0 archiving

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(id_char, scid, ...[, ...])	Create instance from filename parts
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

__init__(**args*, ***kwargs*)

Methods

from_filename_parts(id_char, scid, ...[, ...])	Create instance from filename parts
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

Attributes

archive_prefix	Property that contains the generated prefix for L0 archiving
filename_parts	Property that contains a namespace of filename parts
path	Property containing the file path

static _calculate_applicable_time(*start: datetime*, *end: datetime*)

Based on the start time and end time of a file, returns the applicable time (date)

Parameters

- **start** (*datetime.datetime*) – Start of the applicable time range
- **end** (*datetime.datetime*) – End of the applicable time range

Returns
The date of the mean time between start and end

Return type
datetime.date

classmethod _format_filename_parts(*id_char: str*, *scid: int*, *first_apid: int*, *fill: str*, *created_time: datetime*, *numeric_id: int*, *file_number: int*, *extension: str*, *signal: str | None = None*)

Construct a path from filename parts

Parameters

- **id_char** (*str*) – Either P (for PDS files, Construction Records) or X (for Delivery Records)
- **scid** (*int*) – Spacecraft ID
- **first_apid** (*int*) – First APID in the file
- **fill** (*str*) – Custom string up to 14 characters long
- **created_time** (*datetime.datetime*) – Creation time of the file
- **numeric_id** (*int*) – Data set ID, 0-9, one digit
- **file_number** (*str*) – File number within the data set. Construction records are always file number zero.
- **extension** (*str*) – File name extension. Either PDR or PDS
- **signal** (*Optional[str], Optional*) – Optional signal suffix. Always '.XFR'

Returns
Formatted filename

Return type
str

classmethod _from_filename_parts(*, *basepath: str | Path | S3Path = None*, ***parts: Any*)

Create instance from filename parts.

The part kwarg names are named according to the regex for the file type.

Parameters

- **basepath** (*Union[str, Path, S3Path], Optional*) – Allows prepending a basepath or prefix.
- **parts** (*Any*) – Passed directly to _format_filename_parts. This is a dict of variable kwargs that will differ in each filename class based on the required parts for that particular filename type.

Return type
AbstractValidFilename

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Property that contains the generated prefix for L0 archiving

property filename_parts

Property that contains a namespace of filename parts

classmethod from_filename_parts(*id_char: str*, *scid: int*, *first_apid: int*, *fill: str*, *created_time: datetime*, *numeric_id: int*, *file_number: int*, *extension: str*, *signal: str | None = None*, *basepath: str | Path | S3Path | None = None*)

Create instance from filename parts

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part names are named according to the regex for the file type.

Parameters

- **id_char** (*str*) – Either P (for PDS files, Construction Records) or X (for Delivery Records)
- **scid** (*int*) – Spacecraft ID
- **first_apid** (*int*) – First APID in the file
- **fill** (*str*) – Custom string up to 14 characters long
- **created_time** (*datetime.datetime*) – Creation time of the file
- **numeric_id** (*int*) – Data set ID, 0-9, one digit
- **file_number** (*str*) – File number within the data set. Construction records are always file number zero.

- **extension** (*str*) – File name extension. Either PDR or PDS
- **signal** (*Optional[str]*) – Optional signal suffix. Always '.XFR'
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
L0Filename

generate_prefixed_path(*parent_path: str | Path | S3Path*) → Path | S3Path

Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)

Parameters
parent_path (*Union[str, Path, S3Path]*) – Absolute path to the parent directory or S3 bucket prefix. The generated path prefix is appended to the parent path and followed by the file basename.

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

property path: Path | S3Path

Property containing the file path

regex_match(*path: str | Path | S3Path*)

Parse and validate a given path against class-attribute defined regex

Returns
Match group dict of filename parts

Return type
dict

libera_utils.io.filenaming.LiberaDataProductFilename

class libera_utils.io.filenaming.**LiberaDataProductFilename**(**args*, ***kwargs*)

Bases: *AbstractValidFilename*

Filename validation class for L1B and L2 science products

Attributes

archive_prefix
Property that contains the generated prefix for L1B and L2 archiving

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(data_level, ...[, ...])	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

__init__(**args*, ***kwargs*)

Methods

from_filename_parts(data_level, ...[, ...])	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

Attributes

archive_prefix	Property that contains the generated prefix for L1B and L2 archiving
filename_parts	Property that contains a namespace of filename parts
path	Property containing the file path

static _calculate_applicable_time(*start: datetime*, *end: datetime*)

Based on the start time and end time of a file, returns the applicable time (date)

Parameters

- **start** (*datetime.datetime*) – Start of the applicable time range
- **end** (*datetime.datetime*) – End of the applicable time range

Returns

The date of the mean time between start and end

Return type

datetime.date

classmethod _format_filename_parts(*data_level: str*, *product_name: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*, *extension: str*)

Construct a path from filename parts

Parameters

- **data_level** (*str*) – L1B or L2

- **product_name** (*str*) – Libera instrument, cam or rad for L1B and cloud-fraction etc. for L2. May contain anything except for underscores.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – First timestamp in the SPK
- **utc_end** (*datetime.datetime*) – Last timestamp in the SPK
- **revision** (*datetime.datetime*) – Time when the file was created.
- **extension** (*str*) – File extension (.nc or .h5)

Returns
Formatted filename

Return type
str

classmethod _from_filename_parts(*, *basepath: str | Path | S3Path = None*, ***parts: Any*)

Create instance from filename parts.

The part kwarg names are named according to the regex for the file type.

Parameters

- **basepath** (*Union[str, Path, S3Path], Optional*) – Allows prepending a basepath or prefix.
- **parts** (*Any*) – Passed directly to _format_filename_parts. This is a dict of variable kwargs that will differ in each filename class based on the required parts for that particular filename type.

Return type
AbstractValidFilename

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Property that contains the generated prefix for L1B and L2 archiving

property filename_parts

Property that contains a namespace of filename parts

classmethod from_filename_parts(*data_level: str*, *product_name: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*, *extension: str = 'nc'*, *basepath: str | Path | S3Path | None = None*)

Create instance from filename parts. All keyword arguments other than basepath are required!

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part names are named according to the regex for the file type.

Parameters

- **data_level** (*str*) – L1B or L2 identifying the level of the data product

- **product_name** (*str*) – Product type. e.g. cloud-fraction for L2 or cam for L1B. May contain anything except for underscores.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – First timestamp in the SPK
- **utc_end** (*datetime.datetime*) – Last timestamp in the SPK
- **revision** (*datetime.datetime*) – Time when the file was created.
- **extension** (*str*) – File extension (.nc or .h5)
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
LiberaDataProductFilename

generate_prefixed_path(*parent_path: str | Path | S3Path*) → Path | S3Path

Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)

Parameters
parent_path (*Union[str, Path, S3Path]*) – Absolute path to the parent directory or S3 bucket prefix. The generated path prefix is appended to the parent path and followed by the file basename.

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

property path: Path | S3Path

Property containing the file path

regex_match(*path: str | Path | S3Path*)

Parse and validate a given path against class-attribute defined regex

Returns
Match group dict of filename parts

Return type
dict

libera_utils.io.filenaming.ManifestFilename

class libera_utils.io.filenaming.**ManifestFilename**(**args*, ***kwargs*)

Bases: *AbstractValidFilename*

Class for naming manifest files

Attributes

archive_prefix
Manifests are not archived like data products, but for convenience and ease of debugging they will be kept in the dropbox bucket by input/output and day they were made.

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(manifest_type, ulid_code)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

__init__(**args*, ***kwargs*)

Methods

from_filename_parts(manifest_type, ulid_code)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

Attributes

archive_prefix	Manifests are not archived like data products, but for convenience and ease of debugging they will be kept in the dropbox bucket by input/output and day they were made.
filename_parts	Property that contains a namespace of filename parts
path	Property containing the file path

static _calculate_applicable_time(*start: datetime*, *end: datetime*)

Based on the start time and end time of a file, returns the applicable time (date)

Parameters

- **start** (*datetime.datetime*) – Start of the applicable time range
- **end** (*datetime.datetime*) – End of the applicable time range

Returns

The date of the mean time between start and end

Return type

datetime.date

classmethod _format_filename_parts(*manifest_type:* ManifestType, *ulid_code: ULID*)

Construct a path from filename parts

Parameters

- **manifest_type** (ManifestType) – Input or output
- **ulid_code** (*ulid.ULID*) – ULID code for use in filename parts

Returns
Formatted filename

Return type
str

classmethod _from_filename_parts(*, *basepath: str | Path | S3Path = None*, ***parts: Any*)

Create instance from filename parts.

The part kwarg names are named according to the regex for the file type.

Parameters

- **basepath** (*Union[str, Path, S3Path], Optional*) – Allows prepending a basepath or prefix.
- **parts** (*Any*) – Passed directly to _format_filename_parts. This is a dict of variable kwargs that will differ in each filename class based on the required parts for that particular filename type.

Return type
AbstractValidFilename

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Manifests are not archived like data products, but for convenience and ease of debugging they will be kept in the dropbox bucket by input/output and day they were made. This is used by the step function clean up function in the CDK. # Generate prefix structure # <manifest_type>/<year>/<month>/<day>

property filename_parts

Property that contains a namespace of filename parts

classmethod from_filename_parts(*manifest_type: ManifestType*, *ulid_code: ULID*, *basepath: str | Path | S3Path = None*)

Create instance from filename parts.

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part names are named according to the regex for the file type.

Parameters

- **manifest_type** (ManifestType) – Input or output
- **ulid_code** (*ulid.ULID*) – ULID code for use in filename parts
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
ManifestFilename

generate_prefixed_path(*parent_path: str | Path | S3Path*) → Path | S3Path

Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)

Parameters
parent_path (*Union[str, Path, S3Path]*) – Absolute path to the parent directory or S3 bucket prefix. The generated path prefix is appended to the parent path and followed by the file basename.

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

property path: Path | S3Path

Property containing the file path

regex_match(*path: str | Path | S3Path*)

Parse and validate a given path against class-attribute defined regex

Returns
Match group dict of filename parts

Return type
dict

libera_utils.io.filenaming.ManifestType

class libera_utils.io.filenaming.**ManifestType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enumerated legal manifest type values

__init__(**args*, ***kwds*)

Attributes

INPUT
input
OUTPUT
output

class libera_utils.io.filenaming.**AbstractValidFilename**(**args*, ***kwargs*)

Composition of a CloudPath/Path instance with some methods to perform regex validation on filenames

Attributes

archive_prefix
Property that contains the generated prefix used for archiving, when applicable

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(*args[, basepath])	Abstract method that must be implemented to provide hinting for required parts
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

static _calculate_applicable_time(*start: datetime*, *end: datetime*)

Based on the start time and end time of a file, returns the applicable time (date)

Parameters

- **start** (*datetime.datetime*) – Start of the applicable time range
- **end** (*datetime.datetime*) – End of the applicable time range

Returns
The date of the mean time between start and end

Return type
datetime.date

abstract classmethod _format_filename_parts(***parts*)

Format parts into a filename

Note: When this is implemented by concrete classes, **parts becomes a set of explicitly named arguments

classmethod _from_filename_parts(*, *basepath: str | Path | S3Path = None*, ***parts: Any*)

Create instance from filename parts.

The part kwarg names are named according to the regex for the file type.

Parameters

- **basepath** (*Union[str, Path, S3Path], Optional*) – Allows prepending a basepath or prefix.
- **parts** (*Any*) – Passed directly to _format_filename_parts. This is a dict of variable kwargs that will differ in each filename class based on the required parts for that particular filename type.

Return type
AbstractValidFilename

abstract _parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

abstract property archive_prefix

Property that contains the generated prefix used for archiving, when applicable

property filename_parts

Property that contains a namespace of filename parts

abstract classmethod from_filename_parts(**args: Any*, *basepath: str | Path | S3Path = None*, ***kwargs: Any*)

Abstract method that must be implemented to provide hinting for required parts

generate_prefixed_path(*parent_path: str | Path | S3Path*) → Path | S3Path

Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)

Parameters

parent_path (*Union[str, Path, S3Path]*) – Absolute path to the parent directory or S3 bucket prefix. The generated path prefix is appended to the parent path and followed by the file basename.

Return type

pathlib.Path or cloudpathlib.s3.s3path.S3Path

property path: Path | S3Path

Property containing the file path

regex_match(*path: str | Path | S3Path*)

Parse and validate a given path against class-attribute defined regex

Returns

Match group dict of filename parts

Return type

dict

class libera_utils.io.filenaming.**AnyFilename**(**args*, ***kwargs*)

Polymorphic class for creating a Filename object

class libera_utils.io.filenaming.**AttitudeKernelFilename**(**args*, ***kwargs*)

Class to construct, store, and manipulate an SPK filename

Attributes

archive_prefix

Property that contains the generated prefix for SPICE archiving

filename_parts

Property that contains a namespace of filename parts

path

Property containing the file path

Methods

from_filename_parts(ck_object, version, ...)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

classmethod _format_filename_parts(*ck_object: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*)

Format filename parts as a string

Parameters

- **ck_object** (*str*) – Name of object whose attitude is represented in this CK.
- **utc_start** (*datetime.datetime*) – Start time of data.
- **utc_end** (*datetime.datetime*) – End time of data.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **revision** (*datetime.datetime*) – When the file was last revised.

Return type
str

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Property that contains the generated prefix for SPICE archiving

classmethod from_filename_parts(*ck_object: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*, *basepath: str | Path | S3Path | None = None*)

Create instance from filename parts.

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part arg names are named according to the regex for the file type.

Parameters

- **ck_object** (*str*) – Name of object whose attitude is represented in this CK.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – Start time of data.
- **utc_end** (*datetime.datetime*) – End time of data.
- **revision** (*datetime.datetime*) – When the file was last revised.

- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
AttitudeKernelFilename

class libera_utils.io.filenaming.**DataLevel**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Data product level

class libera_utils.io.filenaming.**EphemerisKernelFilename**(**args*, ***kwargs*)

Class to construct, store, and manipulate an SPK filename

Attributes

archive_prefix
Property that contains the generated prefix for SPICE archiving

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(spk_object, version, ...)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

classmethod **_format_filename_parts**(*spk_object: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*)

Format filename parts as a string

Parameters

- **spk_object** (*str*) – Name of object whose ephemeris is represented in this SPK.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – Start time of data.
- **utc_end** (*datetime.datetime*) – End time of data.
- **revision** (*datetime.datetime*) – Time when the file was last revised

Return type
str

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Property that contains the generated prefix for SPICE archiving

classmethod from_filename_parts(*spk_object: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*, *basepath: str | Path | S3Path | None = None*)

Create instance from filename parts.

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part arg names are named according to the regex for the file type.

Parameters

- **spk_object** (*str*) – Name of object whose attitude is represented in this SPK.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – Start time of data.
- **utc_end** (*datetime.datetime*) – End time of data.
- **revision** (*datetime.datetime*) – When the file was last revised.
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
EphemerisKernelFilename

class libera_utils.io.filenaming.**L0Filename**(**args*, ***kwargs*)

Filename validation class for L0 files from EDOS.

Attributes

archive_prefix
Property that contains the generated prefix for L0 archiving

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(id_char, scid, ...[, ...])	Create instance from filename parts
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

classmethod _format_filename_parts(*id_char: str*, *scid: int*, *first_apid: int*, *fill: str*, *created_time: datetime*, *numeric_id: int*, *file_number: int*, *extension: str*, *signal: str | None = None*)

Construct a path from filename parts

Parameters

- **id_char** (*str*) – Either P (for PDS files, Construction Records) or X (for Delivery Records)
- **scid** (*int*) – Spacecraft ID
- **first_apid** (*int*) – First APID in the file
- **fill** (*str*) – Custom string up to 14 characters long
- **created_time** (*datetime.datetime*) – Creation time of the file
- **numeric_id** (*int*) – Data set ID, 0-9, one digit
- **file_number** (*str*) – File number within the data set. Construction records are always file number zero.
- **extension** (*str*) – File name extension. Either PDR or PDS
- **signal** (*Optional[str], Optional*) – Optional signal suffix. Always '.XFR'

Returns
Formatted filename

Return type
str

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Property that contains the generated prefix for L0 archiving

classmethod from_filename_parts(*id_char: str*, *scid: int*, *first_apid: int*, *fill: str*, *created_time: datetime*, *numeric_id: int*, *file_number: int*, *extension: str*, *signal: str | None = None*, *basepath: str | Path | S3Path | None = None*)

Create instance from filename parts

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part names are named according to the regex for the file type.

Parameters

- **id_char** (*str*) – Either P (for PDS files, Construction Records) or X (for Delivery Records)
- **scid** (*int*) – Spacecraft ID
- **first_apid** (*int*) – First APID in the file
- **fill** (*str*) – Custom string up to 14 characters long
- **created_time** (*datetime.datetime*) – Creation time of the file
- **numeric_id** (*int*) – Data set ID, 0-9, one digit

- **file_number** (*str*) – File number within the data set. Construction records are always file number zero.
- **extension** (*str*) – File name extension. Either PDR or PDS
- **signal** (*Optional[str]*) – Optional signal suffix. Always '.XFR'
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
L0Filename

class libera_utils.io.filenaming.**LiberaDataProductFilename**(**args*, ***kwargs*)

Filename validation class for L1B and L2 science products

Attributes

archive_prefix
Property that contains the generated prefix for L1B and L2 archiving

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(data_level, ...[, ...])	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

classmethod **_format_filename_parts**(*data_level: str*, *product_name: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*, *extension: str*)

Construct a path from filename parts

Parameters

- **data_level** (*str*) – L1B or L2
- **product_name** (*str*) – Libera instrument, cam or rad for L1B and cloud-fraction etc. for L2. May contain anything except for underscores.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – First timestamp in the SPK
- **utc_end** (*datetime.datetime*) – Last timestamp in the SPK
- **revision** (*datetime.datetime*) – Time when the file was created.
- **extension** (*str*) – File extension (.nc or .h5)

Returns
Formatted filename

Return type
str

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Property that contains the generated prefix for L1B and L2 archiving

classmethod from_filename_parts(*data_level: str*, *product_name: str*, *version: str*, *utc_start: datetime*, *utc_end: datetime*, *revision: datetime*, *extension: str = 'nc'*, *basepath: str | Path | S3Path | None = None*)

Create instance from filename parts. All keyword arguments other than basepath are required!

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part names are named according to the regex for the file type.

Parameters

- **data_level** (*str*) – L1B or L2 identifying the level of the data product
- **product_name** (*str*) – Product type. e.g. cloud-fraction for L2 or cam for L1B. May contain anything except for underscores.
- **version** (*str*) – Software version that the file was created with. Corresponds to the algorithm version as determined by the algorithm software.
- **utc_start** (*datetime.datetime*) – First timestamp in the SPK
- **utc_end** (*datetime.datetime*) – Last timestamp in the SPK
- **revision** (*datetime.datetime*) – Time when the file was created.
- **extension** (*str*) – File extension (.nc or .h5)
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
LiberaDataProductFilename

class libera_utils.io.filenaming.**ManifestFilename**(**args*, ***kwargs*)

Class for naming manifest files

Attributes

archive_prefix
Manifests are not archived like data products, but for convenience and ease of debugging they will be kept in the dropbox bucket by input/output and day they were made.

filename_parts
Property that contains a namespace of filename parts

path
Property containing the file path

Methods

from_filename_parts(manifest_type, ulid_code)	Create instance from filename parts.
generate_prefixed_path(parent_path)	Generates an absolute path of the form {parent_path}/{prefix_structure}/{file_basename} The parent_path can be an S3 bucket or an absolute local filepath (must start with /)
regex_match(path)	Parse and validate a given path against class-attribute defined regex

classmethod _format_filename_parts(*manifest_type: ManifestType*, *ulid_code: ULID*)

Construct a path from filename parts

Parameters

- **manifest_type** (ManifestType) – Input or output
- **ulid_code** (*ulid.ULID*) – ULID code for use in filename parts

Returns
Formatted filename

Return type
str

_parse_filename_parts()

Parse the filename parts into objects from regex matched strings

Returns
namespace object containing filename parts as parsed objects

Return type
types.SimpleNamespace

property archive_prefix

Manifests are not archived like data products, but for convenience and ease of debugging they will be kept in the dropbox bucket by input/output and day they were made. This is used by the step function clean up function in the CDK. # Generate prefix structure # <manifest_type>/<year>/<month>/<day>

classmethod from_filename_parts(*manifest_type: ManifestType*, *ulid_code: ULID*, *basepath: str | Path | S3Path = None*)

Create instance from filename parts.

This method exists primarily to expose typehinting to the user for use with the generic _from_filename_parts. The part names are named according to the regex for the file type.

Parameters

- **manifest_type** (ManifestType) – Input or output
- **ulid_code** (*ulid.ULID*) – ULID code for use in filename parts
- **basepath** (*Optional[Union[str, Path, S3Path]]*) – Allows prepending a basepath or prefix.

Return type
ManifestFilename

class libera_utils.io.filenaming.**ManifestType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enumerated legal manifest type values

libera_utils.io.filenaming.**format_semantic_version**(*semantic_version: str*)

Formats a semantic version string X.Y.Z into a filename-compatible string like VX-Y-Z, for X = major version, Y = minor version, Z = patch.

Result is uppercase. Release candidate suffixes are allowed as no strict checking is done on the contents of X, Y, or Z. e.g. 1.2.3rc1 becomes V1-2-3RC1

Parameters
semantic_version (*str*) – String matching X.Y.Z where X, Y and Z are integers of any length

Return type
str

libera_utils.io.filenaming.**get_current_revision_str**()

Get the current *r%y%j%H%M%S* string for filename revisions.

Returns
Current (now) revision string.

Return type
str

libera_utils.io.filenaming.**get_current_version_str**(*package_name: str*)

Retrieve the current version of a (algorithm) package and format it for inclusion in a filename

Parameters
package_name (*str*) – Package for which to retrieve a version string. This should be your algorithm package and it must use a semantic versioning scheme, configured in project metadata.

Returns
Version string in format vM1m2p3

Return type
str

libera_utils.io.hdf

Utils for HDF5 file handling

Functions

h5dump(f[, include_attrs, stdout])	Prints the contents of an HDF5 object.

libera_utils.io.hdf.h5dump

libera_utils.io.hdf.**h5dump**(*f: File*, *include_attrs: bool = True*, *stdout: bool = False*)

Prints the contents of an HDF5 object.

Parameters

- **f** (*h5py.File or h5py.Group*) – File, Group object from which to start inspecting.
- **include_attrs** (*bool, Optional*) – Default True.
- **stdout** (*bool, Optional*) – Default False. If True, prints to stdout as the object tree is traversed.

Returns

Concatenated string of HDF5 contents

Return type

str

libera_utils.io.hdf.**h5dump**(*f: File*, *include_attrs: bool = True*, *stdout: bool = False*)

Prints the contents of an HDF5 object.

Parameters

- **f** (*h5py.File or h5py.Group*) – File, Group object from which to start inspecting.
- **include_attrs** (*bool, Optional*) – Default True.
- **stdout** (*bool, Optional*) – Default False. If True, prints to stdout as the object tree is traversed.

Returns

Concatenated string of HDF5 contents

Return type

str

libera_utils.io.manifest

Module for manifest file handling

Classes

Manifest(manifest_type[, files, ...])	Object representation of a JSON manifest file

libera_utils.io.manifest.Manifest

class libera_utils.io.manifest.**Manifest**(*manifest_type: ManifestType*, *files: list = None*, *configuration: dict = None*, *filename: str = None*)

Bases: object

Object representation of a JSON manifest file

Methods

add_desired_time_range(start_datetime, ...)	Add a file to the manifest from filename
add_file_to_manifest(file)	Deprecated legacy method replaced by add_files
add_files(*files)	Add files to the manifest from filename
from_file(filepath)	Read a manifest file and return a Manifest object (factory method).
output_manifest_from_input_manifest(...)	Create Output manifest from input manifest file path, adds input files to output manifest configuration
to_json_dict()	Create a dict representation suitable for writing out.
validate()	Validate the contents of this manifest object
validate_checksums()	Validate checksums of listed files
write(outpath[, filename])	Write a manifest file from a Manifest object (self).

__init__(*manifest_type:* ManifestType, *files:* *list* = *None*, *configuration:* *dict* = *None*, *filename:* *str* = *None*)

Constructor

Parameters

- **manifest_type** (*ManifestType*) – Type of manifest
- **files** (*list, Optional*) – List of dictionaries. Each entry must contain a *filename* key and a *checksum* key.
- **configuration** (*dict, Optional*) – Freeform dictionary of configuration items. It's up to the consumer to understand this JSON object.
- **filename** (*str or ManifestFilename, Optional*) – Preset filename. Must be a ManifestFilename object or a str representing a valid manifest file path.

Methods

add_desired_time_range(start_datetime, ...)	Add a file to the manifest from filename
add_file_to_manifest(file)	Deprecated legacy method replaced by add_files
add_files(*files)	Add files to the manifest from filename
from_file(filepath)	Read a manifest file and return a Manifest object (factory method).
output_manifest_from_input_manifest(...)	Create Output manifest from input manifest file path, adds input files to output manifest configuration
to_json_dict()	Create a dict representation suitable for writing out.
validate()	Validate the contents of this manifest object
validate_checksums()	Validate checksums of listed files
write(outpath[, filename])	Write a manifest file from a Manifest object (self).

_generate_filename()

Generate a valid manifest filename

add_desired_time_range(*start_datetime:* *datetime*, *end_datetime:* *datetime*)

Add a file to the manifest from filename

Parameters

- **start_datetime** (*datetime.datetime*) – The desired start time for the range of data in this manifest

- **end_datetime** (*datetime.datetime*) – The desired end time for the range of data in this manifest

Return type
None

add_file_to_manifest(*file*)

Deprecated legacy method replaced by add_files

add_files(**files*)

Add files to the manifest from filename

Parameters
files (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to the file to add to the manifest.

Return type
None

classmethod from_file(*filepath: str*)

Read a manifest file and return a Manifest object (factory method).

Parameters
filepath (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Location of manifest file to read.

Return type
Manifest

classmethod output_manifest_from_input_manifest(*input_manifest: Path*) → *Manifest*

Create Output manifest from input manifest file path, adds input files to output manifest configuration

Parameters
input_manifest (*pathlib.Path or cloudpathlib.s3.s3path.S3Path or Manifest*) – An S3 or regular path to an input_manifest object, or the input manifest object itself

Returns
output_manifest – The newly created output manifest

Return type
Manifest

to_json_dict()

Create a dict representation suitable for writing out.

Return type
dict

validate()

Validate the contents of this manifest object

validate_checksums()

Validate checksums of listed files

write(*outpath: str*, *filename: str = None*)

Write a manifest file from a Manifest object (self).

Parameters

- **outpath** (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Directory path to write to (directory being used loosely to refer also to an S3 bucket path).

- **filename** (*str, Optional*) – Optional filename, must be a valid manifest filename. If not provided, the method uses the objects internal filename attribute. If that is not set, then a filename is automatically generated.

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

Exceptions

ManifestError	Generic exception related to manifest file handling

libera_utils.io.manifest.ManifestError

exception libera_utils.io.manifest.**ManifestError**

Generic exception related to manifest file handling

class libera_utils.io.manifest.**Manifest**(*manifest_type: ManifestType*, *files: list = None*, *configuration: dict = None*, *filename: str = None*)

Object representation of a JSON manifest file

Methods

add_desired_time_range(start_datetime, ...)	Add a file to the manifest from filename
add_file_to_manifest(file)	Deprecated legacy method replaced by add_files
add_files(*files)	Add files to the manifest from filename
from_file(filepath)	Read a manifest file and return a Manifest object (factory method).
output_manifest_from_input_manifest(...)	Create Output manifest from input manifest file path, adds input files to output manifest configuration
to_json_dict()	Create a dict representation suitable for writing out.
validate()	Validate the contents of this manifest object
validate_checksums()	Validate checksums of listed files
write(outpath[, filename])	Write a manifest file from a Manifest object (self).

_generate_filename()

Generate a valid manifest filename

add_desired_time_range(*start_datetime: datetime*, *end_datetime: datetime*)

Add a file to the manifest from filename

Parameters

- **start_datetime** (*datetime.datetime*) – The desired start time for the range of data in this manifest
- **end_datetime** (*datetime.datetime*) – The desired end time for the range of data in this manifest

Return type
None

add_file_to_manifest(*file*)

Deprecated legacy method replaced by add_files

add_files(**files*)

Add files to the manifest from filename

Parameters
files (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to the file to add to the manifest.

Return type
None

classmethod from_file(*filepath: str*)

Read a manifest file and return a Manifest object (factory method).

Parameters
filepath (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Location of manifest file to read.

Return type
Manifest

classmethod output_manifest_from_input_manifest(*input_manifest: Path*) → *Manifest*

Create Output manifest from input manifest file path, adds input files to output manifest configuration

Parameters
input_manifest (*pathlib.Path or cloudpathlib.s3.s3path.S3Path or Manifest*) – An S3 or regular path to an input_manifest object, or the input manifest object itself

Returns
output_manifest – The newly created output manifest

Return type
Manifest

to_json_dict()

Create a dict representation suitable for writing out.

Return type
dict

validate()

Validate the contents of this manifest object

validate_checksums()

Validate checksums of listed files

write(*outpath: str*, *filename: str = None*)

Write a manifest file from a Manifest object (self).

Parameters

- **outpath** (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Directory path to write to (directory being used loosely to refer also to an S3 bucket path).
- **filename** (*str, Optional*) – Optional filename, must be a valid manifest filename. If not provided, the method uses the objects internal filename attribute. If that is not set, then a filename is automatically generated.

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

exception libera_utils.io.manifest.**ManifestError**
Generic exception related to manifest file handling

libera_utils.io.smart_open

Module for smart_open

Functions

is_gzip(path)	Determine if a string points to an gzip file.
is_s3(path)	Determine if a string points to an s3 location or not.
smart_copy_file(source_path, dest_path[, delete])	Copy function that can handle local files or files in an S3 bucket.
smart_open(path[, mode, enable_gzip])	Open function that can handle local files or files in an S3 bucket.

libera_utils.io.smart_open.is_gzip

libera_utils.io.smart_open.**is_gzip**(*path: str*)
Determine if a string points to an gzip file.

Parameters
path (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to check.

Return type
bool

libera_utils.io.smart_open.is_s3

libera_utils.io.smart_open.**is_s3**(*path: str*)
Determine if a string points to an s3 location or not.

Parameters
path (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to determine if it is and s3 location or not.

Return type
bool

libera_utils.io.smart_open.smart_copy_file

libera_utils.io.smart_open.**smart_copy_file**(*source_path: str*, *dest_path: str*, *delete=False*)

Copy function that can handle local files or files in an S3 bucket. Returns the path to the newly created file as a Path or an S3Path, depending on the destination.

Parameters

- **source_path** (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to the source file to be copied. Files residing in an s3 bucket must begin with "s3://".
- **dest_path** (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to the Destination file to be copied to. Files residing in an s3 bucket must begin with "s3://".
- **delete** (*bool*) – If true, deletes files copied from source (default = False)

Returns
The path to the newly created file

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

libera_utils.io.smart_open.smart_open

libera_utils.io.smart_open.**smart_open**(*path: str*, *mode: str = 'rb'*, *enable_gzip: bool = True*)

Open function that can handle local files or files in an S3 bucket. It also correctly handles gzip files determined by a **.gz* extension.

Parameters

- **path** (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to the file to be opened. Files residing in an s3 bucket must begin with "s3://".
- **mode** (*str, Optional*) – Optional string specifying the mode in which the file is opened. Defaults to 'rb'.
- **enable_gzip** (*bool, Optional*) – Flag to specify that **.gz* files should be opened as a *GzipFile* object. Setting this to False is useful when creating the md5sum of a **.gz* file. Defaults to True.

Return type
IO or gzip.GzipFile

libera_utils.io.smart_open.**_copy_local_to_local**(*source_path: str*, *dest_path: str*, *delete: bool*)

Copy a local source file to a local destination.

Parameters

- **source_path** (*str or pathlib.Path*) – Path to the source file to be copied.
- **dest_path** (*str or pathlib.Path*) – Path to the destination for the copied file.
- **delete** (*bool*) – If true, deletes files copied from source (default = False)

Returns
The path to the newly created file

Return type
pathlib.Path

libera_utils.io.smart_open.**_copy_local_to_s3**(*source_path: str*, *dest_path: str*, *delete: bool*)

Copy a local file to an S3 object.

Parameters

- **source_path** (*str or pathlib.Path*) – Path to the source file to be copied.
- **dest_path** (*str or cloudpathlib.s3.s3path.S3Path*) – Path to the destination for the copied file. Files residing in an s3 bucket must begin with "s3://".
- **delete** (*bool*) – If true, deletes files copied from source (default = False)

Returns
The path to the newly created file

Return type
cloudpathlib.s3.s3path.S3Path

libera_utils.io.smart_open.**_copy_s3_to_local**(*source_path: str*, *dest_path: str*, *delete: bool*)

Copy an S3 object to a local file.

Parameters

- **source_path** (*str or cloudpathlib.s3.s3path.S3Path*) – Path to the source file to be copied. Files residing in an s3 bucket must begin with "s3://".
- **dest_path** (*str or pathlib.Path*) – Path to the destination for the copied file.
- **delete** (*bool*) – If true, deletes files copied from source (default = False)

Returns
The path to the newly created file

Return type
pathlib.Path

libera_utils.io.smart_open.**_copy_s3_to_s3**(*source_path: str*, *dest_path: str*, *delete: bool*)

Copy an S3 object to a different S3 object.

Parameters

- **source_path** (*str or cloudpathlib.s3.s3path.S3Path*) – Path to the source file to be copied. Files residing in an s3 bucket must begin with "s3://".
- **dest_path** (*str or cloudpathlib.s3.s3path.S3Path*) – Path to the Destination file to be copied to. Files residing in an s3 bucket must begin with "s3://".
- **delete** (*bool*) – If true, deletes files copied from source (default = False)

Returns
The path to the newly created file

Return type
cloudpathlib.s3.s3path.S3Path

libera_utils.io.smart_open.**is_gzip**(*path: str*)

Determine if a string points to an gzip file.

Parameters
path (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to check.

Return type
bool

libera_utils.io.smart_open.**is_s3**(*path: str*)

Determine if a string points to an s3 location or not.

Parameters
path (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to determine if it is and s3 location or not.

Return type
bool

libera_utils.io.smart_open.**smart_copy_file**(*source_path: str*, *dest_path: str*, *delete=False*)

Copy function that can handle local files or files in an S3 bucket. Returns the path to the newly created file as a Path or an S3Path, depending on the destination.

Parameters

- **source_path** (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to the source file to be copied. Files residing in an s3 bucket must begin with "s3://".
- **dest_path** (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to the Destination file to be copied to. Files residing in an s3 bucket must begin with "s3://".
- **delete** (*bool*) – If true, deletes files copied from source (default = False)

Returns
The path to the newly created file

Return type
pathlib.Path or cloudpathlib.s3.s3path.S3Path

libera_utils.io.smart_open.**smart_open**(*path: str*, *mode: str = 'rb'*, *enable_gzip: bool = True*)

Open function that can handle local files or files in an S3 bucket. It also correctly handles gzip files determined by a **.gz* extension.

Parameters

- **path** (*str or pathlib.Path or cloudpathlib.s3.s3path.S3Path*) – Path to the file to be opened. Files residing in an s3 bucket must begin with "s3://".
- **mode** (*str, Optional*) – Optional string specifying the mode in which the file is opened. Defaults to 'rb'.
- **enable_gzip** (*bool, Optional*) – Flag to specify that **.gz* files should be opened as a *GzipFile* object. Setting this to False is useful when creating the md5sum of a **.gz* file. Defaults to True.

Return type
IO or gzip.GzipFile

3.1.7 libera_utils.kernel_maker

Module containing CLI tool for creating SPICE kernels from packets

Functions

get_spice_packet_data_from_filepaths(...)	Utility function to return an array of packet data from a list of file paths of raw JPSS APID 11 geolocation packet data files.
make_azel_ck(parsed_args)	Create a Libera Az-El CK from CCSDS packets or ASCII input files The C-kernel (CK) is the component of SPICE concerned with attitude of spacecraft structures or instruments.
make_jpss_ck(parsed_args)	Create a JPSS CK from APID 11 CCSDS packets.
make_jpss_kernels_from_manifest(...)	Alpha function triggering kernel generation from manifest file.
make_jpss_spk(parsed_args)	Create a JPSS SPK from APID 11 CCSDS packets.
write_kernel_input_file(data, filepath[, ...])	Write ephemeris and attitude data to MKSPK and MSOPCK input data files, respectively.
write_kernel_setup_file(data, filepath)	Write an MSOPCK or MKSPK compatible setup file of key-value pairs.

libera_utils.kernel_maker.get_spice_packet_data_from_filepaths

libera_utils.kernel_maker.**get_spice_packet_data_from_filepaths**(*packet_data_filepaths*)

Utility function to return an array of packet data from a list of file paths of raw JPSS APID 11 geolocation packet data files.

Parameters

packet_data_filepaths
[list] The list of file paths to the raw packet data

:returns: **packet_data – The configured packet data. See packets.py for more details on structure**
:rtype: numpy.ndarray

libera_utils.kernel_maker.make_azel_ck

libera_utils.kernel_maker.**make_azel_ck**(*parsed_args: Namespace*)

Create a Libera Az-El CK from CCSDS packets or ASCII input files The C-kernel (CK) is the component of SPICE concerned with attitude of spacecraft structures or instruments.

Parameters
parsed_args (*argparse.Namespace*) – Namespace of parsed CLI arguments

Return type
None

libera_utils.kernel_maker.make_jpss_ck

libera_utils.kernel_maker.**make_jpss_ck**(*parsed_args: Namespace*)

Create a JPSS CK from APID 11 CCSDS packets. The C-kernel (CK) is the component of SPICE concerned with attitude of spacecraft structures or instruments.

Parameters
parsed_args (*argparse.Namespace*) – Namespace of parsed CLI arguments

Return type
None

libera_utils.kernel_maker.make_jpss_kernels_from_manifest

libera_utils.kernel_maker.**make_jpss_kernels_from_manifest**(*manifest_file_path: str*, *output_directory: str*)

Alpha function triggering kernel generation from manifest file.

If the manifest configuration field contains "start_time" and "end_time" fields then this function will select only packet data that falls in that range. If these are not given, then all packet data will be used.

Parameters

- **manifest_file_path** (*str or cloudpathlib.anypath.AnyPath*) – Path to the manifest file that includes end_time and start_time in the configuration section
- **output_directory** (*str or cloudpathlib.anypath.AnyPath*) – Path to save the completed kernels

Returns
output_directory – Path to the directory containing the completed kernels

Return type
str or cloudpathlib.anypath.AnyPath

libera_utils.kernel_maker.make_jpss_spk

libera_utils.kernel_maker.**make_jpss_spk**(*parsed_args: Namespace*)

Create a JPSS SPK from APID 11 CCSDS packets. The SPK system is the component of SPICE concerned with ephemeris data (position/velocity).

Parameters
parsed_args (*argparse.Namespace*) – Namespace of parsed CLI arguments

Return type
None

libera_utils.kernel_maker.write_kernel_input_file

libera_utils.kernel_maker.**write_kernel_input_file**(*data: ndarray*, *filepath: str*, *fields: list = None*, *fmt: str = '%.16f'*)

Write ephemeris and attitude data to MKSPK and MSOPCK input data files, respectively.

See MSOPCK documentation here:
https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/ug/msopck.html

See MKSPK documentation here:
https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/ug/mkspk.html

Parameters

- **data** (*numpy.ndarray*) – Structured array (named, with data types) of attitude or ephemeris data.
- **filepath** (*str or pathlib.Path*) – Filepath to write to.
- **fields** (*list*) – Optional. List of field names to write out to the data file. If not specified, assume fields are already in the proper order.
- **fmt** (*str or list*) – Format specifier(s) to pass to np.savetxt. Default is to assume everything should be floats with 16 decimal places of precision (%.16f). If a list is passed, it must contain a format specifier for each column in data.

Returns
Absolute path to written file.

Return type
pathlib.Path

libera_utils.kernel_maker.write_kernel_setup_file

libera_utils.kernel_maker.**write_kernel_setup_file**(*data: dict*, *filepath: Path*)

Write an MSOPCK or MKSPK compatible setup file of key-value pairs. See documentation here: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/ug/msopck.html#Input%20Data%20Format

Parameters

- **data** (*dict*) – Dictionary of key-value pairs to write to the setup file.
- **filepath** (*pathlib.Path*) – Filepath to write to.

Returns
Absolute path to written file.

Return type
pathlib.Path

libera_utils.kernel_maker.**get_spice_packet_data_from_filepaths**(*packet_data_filepaths*)

Utility function to return an array of packet data from a list of file paths of raw JPSS APID 11 geolocation packet data files.

Parameters

packet_data_filepaths
[list] The list of file paths to the raw packet data

:returns: **packet_data – The configured packet data. See packets.py for more details on structure**
:rtype: numpy.ndarray

libera_utils.kernel_maker.**make_azel_ck**(*parsed_args: Namespace*)

Create a Libera Az-El CK from CCSDS packets or ASCII input files The C-kernel (CK) is the component of SPICE concerned with attitude of spacecraft structures or instruments.

Parameters
parsed_args (*argparse.Namespace*) – Namespace of parsed CLI arguments

Return type
None

libera_utils.kernel_maker.**make_jpss_ck**(*parsed_args: Namespace*)

Create a JPSS CK from APID 11 CCSDS packets. The C-kernel (CK) is the component of SPICE concerned with attitude of spacecraft structures or instruments.

Parameters
parsed_args (*argparse.Namespace*) – Namespace of parsed CLI arguments

Return type
None

libera_utils.kernel_maker.**make_jpss_kernels_from_manifest**(*manifest_file_path: str*, *output_directory: str*)

Alpha function triggering kernel generation from manifest file.

If the manifest configuration field contains "start_time" and "end_time" fields then this function will select only packet data that falls in that range. If these are not given, then all packet data will be used.

Parameters

- **manifest_file_path** (*str or cloudpathlib.anypath.AnyPath*) – Path to the manifest file that includes end_time and start_time in the configuration section
- **output_directory** (*str or cloudpathlib.anypath.AnyPath*) – Path to save the completed kernels

Returns
output_directory – Path to the directory containing the completed kernels

Return type
str or cloudpathlib.anypath.AnyPath

libera_utils.kernel_maker.**make_jpss_spk**(*parsed_args: Namespace*)

Create a JPSS SPK from APID 11 CCSDS packets. The SPK system is the component of SPICE concerned with ephemeris data (position/velocity).

Parameters
parsed_args (*argparse.Namespace*) – Namespace of parsed CLI arguments

Return type
None

libera_utils.kernel_maker.**write_kernel_input_file**(*data: ndarray*, *filepath: str*, *fields: list = None*, *fmt: str = '%.16f'*)

Write ephemeris and attitude data to MKSPK and MSOPCK input data files, respectively.

See MSOPCK documentation here:
https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/ug/msopck.html

See MKSPK documentation here:
https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/ug/mkspk.html

Parameters

- **data** (*numpy.ndarray*) – Structured array (named, with data types) of attitude or ephemeris data.
- **filepath** (*str or pathlib.Path*) – Filepath to write to.
- **fields** (*list*) – Optional. List of field names to write out to the data file. If not specified, assume fields are already in the proper order.
- **fmt** (*str or list*) – Format specifier(s) to pass to np.savetxt. Default is to assume everything should be floats with 16 decimal places of precision (%.16f). If a list is passed, it must contain a format specifier for each column in data.

Returns
Absolute path to written file.

Return type
pathlib.Path

libera_utils.kernel_maker.**write_kernel_setup_file**(*data: dict*, *filepath: Path*)

Write an MSOPCK or MKSPK compatible setup file of key-value pairs. See documentation here: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/ug/msopck.html#Input%20Data%20Format

Parameters

- **data** (*dict*) – Dictionary of key-value pairs to write to the setup file.
- **filepath** (*pathlib.Path*) – Filepath to write to.

Returns
Absolute path to written file.

Return type
pathlib.Path

3.1.8 libera_utils.logutil

Logging utilities

Functions

configure_static_logging(config_file)	Configure logging based on a static logging configuration yaml file.
configure_task_logging(task_id[, ...])	Configure logging for a specific task (e.g. a processing algorithm).
flush_cloudwatch_logs()	Force flush of all cloudwatch logging handlers.

libera_utils.logutil.configure_static_logging

libera_utils.logutil.**configure_static_logging**(*config_file: str | Path | S3Path*)

Configure logging based on a static logging configuration yaml file.

The yaml is interpreted as a dict configuration. There is no ability to customize this logging configuration at runtime.

Parameters

config_file (*cloudpathlib.anypath.AnyPath or str*) – Location of config file.

See also:

configure_task_logging

Runtime modifiable logging configuration.

libera_utils.logutil.configure_task_logging

libera_utils.logutil.**configure_task_logging**(*task_id: str*, *limit_debug_loggers: Iterable[str] | str | None = None*, *console_log_level: str | int = 20*, *console_log_json: bool = False*, *log_dir: str | Path | S3Path | None = None*, *cloudwatch_log_group: str | None = None*)

Configure logging for a specific task (e.g. a processing algorithm).

File-based logging is always done at the DEBUG level. Watchtower-based cloudwatch logging is always done at the DEBUG level. Console logging level defaults to INFO but can be set with console_log_level.

Examples

Example 1: The following will configure DEBUG console-only logging for anything in your script but all other loggers will be limited to INFO level.

`python configure_task_logging("my-script", limit_debug_loggers=("__main__",), console_log_level=logging.DEBUG) `

Example 2: This will allow all debug messages through from all loggers and sets up file-based logging and a custom cloudwatch log group. Also console messages will be logged in serialized JSON.

```python configure_task_logging(“my-script”,

console_log_level=logging.DEBUG, log_dir=Path(“/tmp/my-script”), console_log_json=True, cloudwatch_log_group=”custom-log-group”)

```

Parameters

- **task_id** (*str*) – Unique identifier by which to name the log file and cloudwatch log stream.
- **limit_debug_loggers** (*Optional[Union[Iterable[str] | str]]*) – A list of logger name prefixes from which you want to allow debug messages (blocks debug from all others). For example, if you are working on a package called *my_app* and using module level logging, all your loggers will be named like *my_app.module_name.submodule_name*. By setting this to *(my_app,)*, all loggers that are named *my_app.** will propagate debug messages while preventing spammy debug messages from installed libraries like boto3. If this is empty or None, all debug messages will propagate. To use this in scripts, either leave it unset or use *limit_debug_loggers=(“__main__,)*.

- **console_log_level** (*str or int, Optional*) – Log level for console logging. If not specified, defaults to INFO
- **console_log_json** (*bool, Optional*) – If True, console logs will be JSON formatted. This is suitable for setting up loggers in AWS services that are automatically monitored by cloudwatch on stdout and stderr (e.g. Lambda or Batch)
- **log_dir** (*str or Path or S3Path, Optional*) – Log directory, which may be a local or S3Path. Default is None and results in no file-based logging.
- **cloudwatch_log_group** (*str, Optional*) – Override optional environment variable log group name. Default is None and will result in falling back to the LIBERA_LOG_GROUP environment variable. If that is not set, no cloudwatch JSON logging will be configured.

Notes

Even in the absence of cloudwatch JSON logging, all stdout/stderr messages generated by a Lambda will be logged to CloudWatch as string messages. Embedded JSON strings in log message text can still be queried in CloudWatch.

See also:

configure_static_logging
Static logging configuration based on yaml file.

libera_utils.logutil.flush_cloudwatch_logs

libera_utils.logutil.**flush_cloudwatch_logs**()

Force flush of all cloudwatch logging handlers.

If you are missing the last few log messages in a log stream, this may help get those logs ingested before the process shuts down the logging system.

Return type
None

Classes

JsonLogFormatter(*args[, ...])	Altered version of the CloudWatchLogFormatter provided in the watchtower library

libera_utils.logutil.JsonLogFormatter

class libera_utils.logutil.**JsonLogFormatter**(**args*, *add_log_record_attrs: Tuple[str, ...] | None = None*, *add_asctime: bool = True*, ***kwargs*)

Bases: Formatter

Altered version of the CloudWatchLogFormatter provided in the watchtower library

Methods

converter([seconds])	Convert seconds since the Epoch to a time tuple expressing local time.
format(record)	Format log message to a string
formatException(ei)	Format and return the specified exception information as a string.
formatStack(stack_info)	This method is provided as an extension point for specialized formatting of stack information.
formatTime(record[, datefmt])	Return the creation time of the specified LogRecord as formatted text.
usesTime()	Check if the format uses the creation time of the record.

formatMessage

__init__(**args*, *add_log_record_attrs: Tuple[str, ...] | None = None*, *add_asctime: bool = True*, ***kwargs*)

Parameters

- **add_log_record_attrs** (*Optional, tuple*) – Tuple of log record attributes to add to the resulting structured JSON structure that comes out of the logging formatter.
- **add_asctime** (*bool*) – If True, adds an ASCII (ISO 8601-like) timestamp to the log record. Default True.

Methods

format(record)	Format log message to a string

Attributes

default_msec_format
default_time_format

format(*record: LogRecord*) → str

Format log message to a string

Parameters

record (*logging.LogRecord*) – Log record object containing the logged message, which may be a dict (Mapping) or a string

class libera_utils.logutil.**JsonLogFormatter**(**args*, *add_log_record_attrs: Tuple[str, ...] | None = None*, *add_asctime: bool = True*, ***kwargs*)

Altered version of the CloudWatchLogFormatter provided in the watchtower library

Methods

converter([seconds])	Convert seconds since the Epoch to a time tuple expressing local time.
format(record)	Format log message to a string
formatException(ei)	Format and return the specified exception information as a string.
formatStack(stack_info)	This method is provided as an extension point for specialized formatting of stack information.
formatTime(record[, datefmt])	Return the creation time of the specified LogRecord as formatted text.
usesTime()	Check if the format uses the creation time of the record.

formatMessage

format(*record: LogRecord*) → str

Format log message to a string

Parameters

record (*logging.LogRecord*) – Log record object containing the logged message, which may be a dict (Mapping) or a string

libera_utils.logutil.**_json_serialize_default**(*o: Any*) → str

A standard 'default' json serializer function.

- Serializes datetime objects using their .isoformat() method.
- Serializes all other objects using repr().

libera_utils.logutil.**configure_static_logging**(*config_file: str | Path | S3Path*)

Configure logging based on a static logging configuration yaml file.

The yaml is interpreted as a dict configuration. There is no ability to customize this logging configuration at runtime.

Parameters

config_file (*cloudpathlib.anypath.AnyPath or str*) – Location of config file.

See also:

configure_task_logging

Runtime modifiable logging configuration.

libera_utils.logutil.**configure_task_logging**(*task_id: str*, *limit_debug_loggers: Iterable[str] | str | None = None*, *console_log_level: str | int = 20*, *console_log_json: bool = False*, *log_dir: str | Path | S3Path | None = None*, *cloudwatch_log_group: str | None = None*)

Configure logging for a specific task (e.g. a processing algorithm).

File-based logging is always done at the DEBUG level. Watchtower-based cloudwatch logging is always done at the DEBUG level. Console logging level defaults to INFO but can be set with console_log_level.

Examples

Example 1: The following will configure DEBUG console-only logging for anything in your script but all other loggers will be limited to INFO level.

`python configure_task_logging("my-script", limit_debug_loggers=("__main__",), console_log_level=logging.DEBUG) `

Example 2: This will allow all debug messages through from all loggers and sets up file-based logging and a custom cloudwatch log group. Also console messages will be logged in serialized JSON.

```python configure_task_logging("my-script",

    console_log_level=logging.DEBUG, log_dir=Path("/tmp/my-script"), console_log_json=True, cloudwatch_log_group="custom-log-group")

```

Parameters

- **task_id** (*str*) – Unique identifier by which to name the log file and cloudwatch log stream.
- **limit_debug_loggers** (*Optional[Union[Iterable[str] | str]]*) – A list of logger name prefixes from which you want to allow debug messages (blocks debug from all others). For example, if you are working on a package called *my_app* and using module level logging, all your loggers will be named like *my_app.module_name.submodule_name*. By setting this to *(my_app,)*, all loggers that are named *my_app.** will propagate debug messages while preventing spammy debug messages from installed libraries like boto3. If this is empty or None, all debug messages will propagate. To use this in scripts, either leave it unset or use *limit_debug_loggers=("__main__,)*.
- **console_log_level** (*str or int, Optional*) – Log level for console logging. If not specified, defaults to INFO
- **console_log_json** (*bool, Optional*) – If True, console logs will be JSON formatted. This is suitable for setting up loggers in AWS services that are automatically monitored by cloudwatch on stdout and stderr (e.g. Lambda or Batch)
- **log_dir** (*str or Path or S3Path, Optional*) – Log directory, which may be a local or S3Path. Default is None and results in no file-based logging.
- **cloudwatch_log_group** (*str, Optional*) – Override optional environment variable log group name. Default is None and will result in falling back to the LIBERA_LOG_GROUP environment variable. If that is not set, no cloudwatch JSON logging will be configured.

Notes

Even in the absence of cloudwatch JSON logging, all stdout/stderr messages generated by a Lambda will be logged to CloudWatch as string messages. Embedded JSON strings in log message text can still be queried in CloudWatch.

See also:

configure_static_logging
: Static logging configuration based on yaml file.

libera_utils.logutil.**flush_cloudwatch_logs**()

Force flush of all cloudwatch logging handlers.

If you are missing the last few log messages in a log stream, this may help get those logs ingested before the process shuts down the logging system.

Return type
None

3.1.9 libera_utils.packets

Module for reading packet data

Functions

array_from_packets(packets[, apid])	Create an array from a list of packets.
parse_packets(packet_parser, ...[, apid])	Parse a recarray from a list of packet filepaths, assuming the same parser for all

libera_utils.packets.array_from_packets

libera_utils.packets.**array_from_packets**(*packets: list*, *apid: int = None*)

Create an array from a list of packets. This function assumes that the fields and format for every packet is identical for a given APID.

Parameters

- **packets** (*list*) – List of lasp_packets.parser.Packet objects.
- **apid** (*int*) – Application Packet ID to create an array from. We can only create an array for a single APID because we need to assume the same fields in every packet. If not specified, every packet must be of the same APID.

Returns
Record array with one column per field name in the packet type. Values are derived if a derived value exists, otherwise, the values are the raw values.

Return type
numpy.recarray

libera_utils.packets.parse_packets

libera_utils.packets.**parse_packets**(*packet_parser: PacketParser*, *packet_data_filepaths: list*, *apid: int = None*)

Parse a recarray from a list of packet filepaths, assuming the same parser for all

Parameters

- **packet_parser** (*space_packet_parser.parser.PacketParser*) – Parser, already initialized with the anticipated definition.
- **packet_data_filepaths** (*list*) – List of filepaths to packets files.

- **apid** (*int*) – Filter on APID so we don't get mismatches in case the parser finds multiple parsable packet definitions in the files. This can happen if the XTCE document contains definitions for multiple packet types and >1 of those packet types is present in the packet data files.

Returns
Concatenated arrays of packet data.

Return type
numpy.recarray

libera_utils.packets.**array_from_packets**(*packets: list*, *apid: int = None*)

Create an array from a list of packets. This function assumes that the fields and format for every packet is identical for a given APID.

Parameters

- **packets** (*list*) – List of lasp_packets.parser.Packet objects.
- **apid** (*int*) – Application Packet ID to create an array from. We can only create an array for a single APID because we need to assume the same fields in every packet. If not specified, every packet must be of the same APID.

Returns
Record array with one column per field name in the packet type. Values are derived if a derived value exists, otherwise, the values are the raw values.

Return type
numpy.recarray

libera_utils.packets.**parse_packets**(*packet_parser: PacketParser*, *packet_data_filepaths: list*, *apid: int = None*)

Parse a recarray from a list of packet filepaths, assuming the same parser for all

Parameters

- **packet_parser** (*space_packet_parser.parser.PacketParser*) – Parser, already initialized with the anticipated definition.
- **packet_data_filepaths** (*list*) – List of filepaths to packets files.
- **apid** (*int*) – Filter on APID so we don't get mismatches in case the parser finds multiple parsable packet definitions in the files. This can happen if the XTCE document contains definitions for multiple packet types and >1 of those packet types is present in the packet data files.

Returns
Concatenated arrays of packet data.

Return type
numpy.recarray

3.1.10 libera_utils.quality_flags

Quality flag definitions

Functions

with_all_none(f)	Decorator that adds *NONE* and *ALL* psuedo-members to a QualityFlag *f*

libera_utils.quality_flags.with_all_none

libera_utils.quality_flags.**with_all_none**(*f*)

Decorator that adds *NONE* and *ALL* psuedo-members to a QualityFlag *f*

For example:

```
@with_all_none
class MyQualityFlag(QualityFlag, metaclass=FrozenFlagMeta):
    MISSING_DATA = FlagBit(0b1, message="Data is missing!")
    VOLTAGE_TOO_HIGH = FlagBit(0b10, message="Voltage is too high!")
qf = MyQualityFlag.ALL  # Equivalent to MyQualityFlag(0b11)
qf.summary
```

Classes

FlagBit(*args[, message])	Subclass of int to capture both an integer value and an accompanying message
FrozenFlagMeta(name, bases, classdict)	Metaclass that freezes an enum entirely, preventing values from being updated, added, or deleted.
QualityFlag(value[, names, module, ...])	Subclass of Flag that add a method for decomposing a flag into its individual components and a property to return a list of all messages associated with a quality flag

libera_utils.quality_flags.FlagBit

class libera_utils.quality_flags.**FlagBit**(**args*, *message=None*, ***kwargs*)

Bases: int

Subclass of int to capture both an integer value and an accompanying message

Attributes

denominator
the denominator of a rational number in lowest terms

imag
the imaginary part of a complex number

numerator
the numerator of a rational number in lowest terms

real
the real part of a complex number

Methods

as_integer_ratio(/)	Return integer ratio.
bit_count(/)	Number of ones in the binary representation of the absolute value of self.
bit_length(/)	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes(/, bytes[, byteorder, signed])	Return the integer represented by the given array of bytes.
to_bytes(/[, length, byteorder, signed])	Return an array of bytes representing an integer.

__init__(**args*, ***kwargs*)

Methods

as_integer_ratio(/)	Return integer ratio.
bit_count(/)	Number of ones in the binary representation of the absolute value of self.
bit_length(/)	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes(/, bytes[, byteorder, signed])	Return the integer represented by the given array of bytes.
to_bytes(/[, length, byteorder, signed])	Return an array of bytes representing an integer.

Attributes

denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

as_integer_ratio(/)

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count(/)

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length(/)

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

denominator

the denominator of a rational number in lowest terms

from_bytes(/, *bytes*, *byteorder='big'*, *, *signed=False*)

Return the integer represented by the given array of bytes.

bytes
Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Indicates whether two's complement is used to represent the integer.

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

to_bytes(/, *length=1*, *byteorder='big'*, *, *signed=False*)

Return an array of bytes representing an integer.

length
Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

libera_utils.quality_flags.FrozenFlagMeta

class libera_utils.quality_flags.**FrozenFlagMeta**(*name*, *bases*, *classdict*)

Bases: EnumType

Metaclass that freezes an enum entirely, preventing values from being updated, added, or deleted.

Methods

__call__(value[, names, module])	Either returns an existing member, or creates a new enum class.
mro(/)	Return a type's method resolution order.

__init__(**args*, ***kwargs*)

Methods

mro(/)	Return a type's method resolution order.

mro(/)
Return a type's method resolution order.

libera_utils.quality_flags.QualityFlag

class libera_utils.quality_flags.**QualityFlag**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Flag

Subclass of Flag that add a method for decomposing a flag into its individual components and a property to return a list of all messages associated with a quality flag

__init__(**args*, ***kwds*)

class libera_utils.quality_flags.**FlagBit**(**args*, *message=None*, ***kwargs*)

Subclass of int to capture both an integer value and an accompanying message

Attributes

denominator
the denominator of a rational number in lowest terms

imag
: the imaginary part of a complex number

numerator
: the numerator of a rational number in lowest terms

real
: the real part of a complex number

Methods

as_integer_ratio(/)	Return integer ratio.
bit_count(/)	Number of ones in the binary representation of the absolute value of self.
bit_length(/)	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes(/, bytes[, byteorder, signed])	Return the integer represented by the given array of bytes.
to_bytes(/[, length, byteorder, signed])	Return an array of bytes representing an integer.

class libera_utils.quality_flags.**FrozenFlagMeta**(*name*, *bases*, *classdict*)

Metaclass that freezes an enum entirely, preventing values from being updated, added, or deleted.

Methods

__call__(value[, names, module])	Either returns an existing member, or creates a new enum class.
mro(/)	Return a type's method resolution order.

class libera_utils.quality_flags.**QualityFlag**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Subclass of Flag that add a method for decomposing a flag into its individual components and a property to return a list of all messages associated with a quality flag

libera_utils.quality_flags.**with_all_none**(*f*)

Decorator that adds *NONE* and *ALL* psuedo-members to a QualityFlag *f*

For example:

```
@with_all_none
class MyQualityFlag(QualityFlag, metaclass=FrozenFlagMeta):
    MISSING_DATA = FlagBit(0b1, message="Data is missing!")
    VOLTAGE_TOO_HIGH = FlagBit(0b10, message="Voltage is too high!")
qf = MyQualityFlag.ALL  # Equivalent to MyQualityFlag(0b11)
qf.summary
```

3.1.11 libera_utils.spice_utils

Modules for SPICE kernel creation, management, and usage

Functions

ensure_spice([f_py, time_kernels_only])	Before trying to understand this piece of code, read this: https://stackoverflow.com/questions/5929107/decorators-with-parameters/60832711#60832711
find_most_recent_naif_kernel(naif_base_url, ...)	Retrieves the name of the most recent kernel at NAIF.
ls_kernel_coverage(kernel_type[, verbose])	List time coverage of all furnished kernels of a given type
ls_kernels([verbose, log])	List all furnished spice kernels.
ls_spice_constants([verbose])	List all constants in the Spice constant pool

libera_utils.spice_utils.ensure_spice

libera_utils.spice_utils.**ensure_spice**(*f_py: Callable = None*, *time_kernels_only: bool = False*)

Before trying to understand this piece of code, read this: https://stackoverflow.com/questions/5929107/decorators-with-parameters/60832711#60832711

Decorator/wrapper that tries to ensure that a metakernel is furnished in as complete a way as possible.

Control flow overview:

1. **Try simply calling the wrapped function naively.**
   - SUCCESS? Great! We're done.
   - SpiceyError? Go to step 2.
2. **Furnish metakernel at SPICE_METAKERNEL**
   - SUCCESS? Great, return the original function again (so it can be re-run).
   - KeyError? Seems like SPICE_METAKERNEL isn't set, no problem. Go to step 3.

Usage:

Three ways to use this object

1. A decorator with no arguments

```
@ensure_spice
def my_spicey_func(a, b):
    pass
```

2. A decorator with parameters. This is useful if we only need the latest SCLK and LSK kernels for the function involved.

```
@ensure_spice(time_kernels_only=True)
def my_spicey_time_func(a, b):
    pass
```

3. An explicit wrapper function, providing a dynamically set value for parameters, e.g. time_kernels_only

```
wrapped = ensure_spice(spicey_func, time_kernels_only=True)
result = wrapped(*args, **kwargs)
```

Parameters

- **f_py** (*Callable*) – The function requiring SPICE that we are going to wrap if being used explicitly, Otherwise None, in which case ensure_spice is being used, not as a function wrapper (see l2a_processing.py) but as a true decorator without an explicit function argument.
- **time_kernels_only** (*bool, Optional*) – Specify that we only need to furnish time kernels (if SPICE_METAKERNEL is set, we still just furnish that metakernel and assume the time kernels are included.

Returns
Decorated function, with spice error handling

Return type
Callable

libera_utils.spice_utils.find_most_recent_naif_kernel

libera_utils.spice_utils.**find_most_recent_naif_kernel**(*naif_base_url: str*, *kernel_file_regex: str*, *allowed_attempts: int = 3*) → str

Retrieves the name of the most recent kernel at NAIF.

Parameters

- **naif_base_url** (*str*) – URL to search for filenames matching kernel_file_regex
- **kernel_file_regex** (*str*) – Regular expression to match filenames on the naif website
- **allowed_attempts** (*int, Optional*) – Number of allowed download times for naif page default = 3

Returns
Returns the file name of the latest kernel on the naif page (e.g., “naif0012.tls”)

Return type
str

libera_utils.spice_utils.ls_kernel_coverage

libera_utils.spice_utils.**ls_kernel_coverage**(*kernel_type: str*, *verbose: bool = False*) → dict

List time coverage of all furnished kernels of a given type

Parameters

- **kernel_type** (*str*) – Either ‘CK’ or ‘SPK’
- **verbose** (*bool*) – If True, print to stdout also

Returns
Key is filename, value is a list of tuples giving the start and end times in ET.

Return type
dict

libera_utils.spice_utils.ls_kernels

libera_utils.spice_utils.**ls_kernels**(*verbose: bool = False*, *log: bool = False*) → list

List all furnished spice kernels.

Parameters

- **verbose** (*bool*) – If True, print to stdout also
- **log** (*bool*) – Whether or not to log the current kernel pool (this gets called a lot)

Returns
A list of KernelFileRecord named tuples.

Return type
list

libera_utils.spice_utils.ls_spice_constants

libera_utils.spice_utils.**ls_spice_constants**(*verbose: bool = False*) → dict

List all constants in the Spice constant pool

Parameters
verbose – If true, print to stdout also

Returns
Dictionary of kernel constants

Return type
dict

Classes

KernelFileCache(kernel_url[, max_cache_age, ...])	Class for downloading, caching, and furnishing SPICE kernel files locally.
KernelFileRecord(kernel_type, file_name)	Tuple for keeping track of kernel files with default kernel_level
SpiceBody(value[, names, module, qualname, ...])	Enum containing SPICE IDs for ephemeris bodies that we use.
SpiceFrame(value[, names, module, qualname, ...])	Enum containing SPICE IDs for reference frames, possibly defined in the Frame Kernel (FK)
SpiceId(strid, numid)	Class that represents a unique identifier in the NAIF SPICE library
SpiceInstrument(value[, names, module, ...])	Enum containing SPICE IDs for instrument geometries configured in the Instrument Kernel (IK)

libera_utils.spice_utils.KernelFileCache

class libera_utils.spice_utils.**KernelFileCache**(*kernel_url: str*, *max_cache_age: timedelta = datetime.timedelta(days=1)*, *fallback_kernel: Path = None*)

Bases: object

Class for downloading, caching, and furnishing SPICE kernel files locally.

It attempts to find a cached kernel file in the user's cache directory (OS-specific location). If that file is not there or is old, it attempts to download it from the specified location. If it is unable to do that, it can optionally read a fallback file included in the libera_utils package but this is not recommended.

Attributes

cache_dir
Property that calls out to get the proper local cache directory

kernel_basename
Base filename of the kernel.

kernel_path
Return the local path location of the kernel if it exists.

Methods

clear()	Remove cached kernel file
download_kernel(kernel_url[, allowed_attempts])	Downloads a kernel from a URL or an S3 location to the system cache location.
furnsh()	Furnish the cached kernel
is_cached([include_stale])	Check the cache directory for kernel file that is within cache age limit.

__init__(*kernel_url: str*, *max_cache_age: timedelta = datetime.timedelta(days=1)*, *fallback_kernel: Path = None*)

Create a new file cache. Downloading is done on first access of kernel_path if the file is not already cached. Fallback occurs only after failing to download. :param kernel_url: Location of kernel file as a URL or an S3Path :type kernel_url: str or cloudpathlib.S3Path :param max_cache_age: Length of time to tolerate stale kernels in the cache without forcing a redownload. :type max_cache_age: datetime.timedelta :param fallback_kernel: Path pointing to a fallback kernel location. May be None, which disallows a fallback. :type fallback_kernel: pathlib.Path

Methods

clear()	Remove cached kernel file
download_kernel(kernel_url[, allowed_attempts])	Downloads a kernel from a URL or an S3 location to the system cache location.
furnsh()	Furnish the cached kernel
is_cached([include_stale])	Check the cache directory for kernel file that is within cache age limit.

Attributes

cache_dir	Property that calls out to get the proper local cache directory
kernel_basename	Base filename of the kernel.
kernel_path	Return the local path location of the kernel if it exists.

property cache_dir

Property that calls out to get the proper local cache directory

Returns
Path to the proper local cache for the system.

Return type
pathlib.Path

clear()

Remove cached kernel file

download_kernel(*kernel_url: str*, *allowed_attempts: int = 3*) → Path

Downloads a kernel from a URL or an S3 location to the system cache location.

Parameters

- **kernel_url** (*str*) – Filename of kernel on NAIF site, as discovered by find_most_recent_naif_kernel
- **allowed_attempts** (*int, Optional*) – Number of allowed download times for naif kernel default = 3

Returns
Location of downloaded file

Return type
pathlib.Path

furnsh()

Furnish the cached kernel

is_cached(*include_stale: bool = False*) → bool

Check the cache directory for kernel file that is within cache age limit. If present, return True.

Parameters
include_stale (*bool*) – Default False. If True, results include kernel that are past the max age.

Returns
Returns True if kernel is present locally and within the age limit.

Return type
bool

property kernel_basename

Base filename of the kernel.

Return type
str

property kernel_path: Path

Return the local path location of the kernel if it exists. If not, try downloading it. If that fails, return the fallback kernel, if allowed.

libera_utils.spice_utils.KernelFileRecord

class libera_utils.spice_utils.**KernelFileRecord**(*kernel_type: str*, *file_name: str*)

Bases: NamedTuple

Tuple for keeping track of kernel files with default kernel_level

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

__init__(**args*, ***kwargs*)

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

Attributes

file_name	Alias for field number 1
kernel_type	Alias for field number 0

count(*value*, */*)

Return number of occurrences of value.

file_name: str

Alias for field number 1

index(*value*, *start=0*, *stop=sys.maxsize*, */*)

Return first index of value.

Raises ValueError if the value is not present.

kernel_type: str

Alias for field number 0

libera_utils.spice_utils.SpiceBody

class libera_utils.spice_utils.**SpiceBody**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum containing SPICE IDs for ephemeris bodies that we use.

__init__(**args*, ***kwds*)

Attributes

JPSS
SSB
SUN
EARTH
EARTH_MOON_BARYCENTER

libera_utils.spice_utils.SpiceFrame

class libera_utils.spice_utils.**SpiceFrame**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum containing SPICE IDs for reference frames, possibly defined in the Frame Kernel (FK)

__init__(**args*, ***kwds*)

Attributes

J2000
ITRF93
EARTH_FIXED

libera_utils.spice_utils.SpiceId

class libera_utils.spice_utils.**SpiceId**(*strid: str*, *numid: int*)

Bases: NamedTuple

Class that represents a unique identifier in the NAIF SPICE library

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

__init__(**args*, ***kwargs*)

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

Attributes

numid	Alias for field number 1
strid	Alias for field number 0

count(*value*, */*)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=sys.maxsize*, */*)

Return first index of value.

Raises ValueError if the value is not present.

numid: int

Alias for field number 1

strid: str

Alias for field number 0

libera_utils.spice_utils.SpiceInstrument

class libera_utils.spice_utils.**SpiceInstrument**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum containing SPICE IDs for instrument geometries configured in the Instrument Kernel (IK)

__init__(*args, **kwds)

class libera_utils.spice_utils.KernelFileCache(kernel_url: str, max_cache_age: timedelta = datetime.timedelta(days=1), fallback_kernel: Path = None)

Class for downloading, caching, and furnishing SPICE kernel files locally.

It attempts to find a cached kernel file in the user's cache directory (OS-specific location). If that file is not there or is old, it attempts to download it from the specified location. If it is unable to do that, it can optionally read a fallback file included in the libera_utils package but this is not recommended.

Attributes

cache_dir
Property that calls out to get the proper local cache directory

kernel_basename
Base filename of the kernel.

kernel_path
Return the local path location of the kernel if it exists.

Methods

clear()	Remove cached kernel file
download_kernel(kernel_url[, allowed_attempts])	Downloads a kernel from a URL or an S3 location to the system cache location.
furnsh()	Furnish the cached kernel
is_cached([include_stale])	Check the cache directory for kernel file that is within cache age limit.

property cache_dir

Property that calls out to get the proper local cache directory

Returns
Path to the proper local cache for the system.

Return type
pathlib.Path

clear()

Remove cached kernel file

download_kernel(kernel_url: str, allowed_attempts: int = 3) → Path

Downloads a kernel from a URL or an S3 location to the system cache location.

Parameters

- **kernel_url** (*str*) – Filename of kernel on NAIF site, as discovered by find_most_recent_naif_kernel
- **allowed_attempts** (*int, Optional*) – Number of allowed download times for naif kernel default = 3

Returns
Location of downloaded file

Return type
pathlib.Path

furnsh()

Furnish the cached kernel

is_cached(*include_stale: bool = False*) → bool

Check the cache directory for kernel file that is within cache age limit. If present, return True.

Parameters
include_stale (*bool*) – Default False. If True, results include kernel that are past the max age.

Returns
Returns True if kernel is present locally and within the age limit.

Return type
bool

property kernel_basename

Base filename of the kernel.

Return type
str

property kernel_path: Path

Return the local path location of the kernel if it exists. If not, try downloading it. If that fails, return the fallback kernel, if allowed.

class libera_utils.spice_utils.**KernelFileRecord**(*kernel_type: str*, *file_name: str*)

Tuple for keeping track of kernel files with default kernel_level

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

file_name: str

Alias for field number 1

kernel_type: str

Alias for field number 0

class libera_utils.spice_utils.**SpiceBody**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum containing SPICE IDs for ephemeris bodies that we use.

class libera_utils.spice_utils.**SpiceFrame**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum containing SPICE IDs for reference frames, possibly defined in the Frame Kernel (FK)

class libera_utils.spice_utils.**SpiceId**(*strid: str*, *numid: int*)

Class that represents a unique identifier in the NAIF SPICE library

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

numid: int

Alias for field number 1

strid: str

Alias for field number 0

class libera_utils.spice_utils.**SpiceInstrument**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Enum containing SPICE IDs for instrument geometries configured in the Instrument Kernel (IK)

libera_utils.spice_utils.**ensure_spice**(*f_py: Callable = None*, *time_kernels_only: bool = False*)

Before trying to understand this piece of code, read this: https://stackoverflow.com/questions/5929107/decorators-with-parameters/60832711#60832711

Decorator/wrapper that tries to ensure that a metakernel is furnished in as complete a way as possible.

Control flow overview:

1. **Try simply calling the wrapped function naively.**
   - SUCCESS? Great! We're done.
   - SpiceyError? Go to step 2.
2. **Furnish metakernel at SPICE_METAKERNEL**
   - SUCCESS? Great, return the original function again (so it can be re-run).
   - KeyError? Seems like SPICE_METAKERNEL isn't set, no problem. Go to step 3.

Usage:

Three ways to use this object

1. A decorator with no arguments

```
@ensure_spice
def my_spicey_func(a, b):
    pass
```

2. A decorator with parameters. This is useful if we only need the latest SCLK and LSK kernels for the function involved.

```
@ensure_spice(time_kernels_only=True)
def my_spicey_time_func(a, b):
    pass
```

3. An explicit wrapper function, providing a dynamically set value for parameters, e.g. time_kernels_only

```
wrapped = ensure_spice(spicey_func, time_kernels_only=True)
result = wrapped(*args, **kwargs)
```

Parameters

- **f_py** (*Callable*) – The function requiring SPICE that we are going to wrap if being used explicitly, Otherwise None, in which case ensure_spice is being used, not as a function wrapper (see l2a_processing.py) but as a true decorator without an explicit function argument.
- **time_kernels_only** (*bool, Optional*) – Specify that we only need to furnish time kernels (if SPICE_METAKERNEL is set, we still just furnish that metakernel and assume the time kernels are included.

Returns
Decorated function, with spice error handling

Return type
Callable

libera_utils.spice_utils.**find_most_recent_naif_kernel**(*naif_base_url: str*, *kernel_file_regex: str*, *allowed_attempts: int = 3*) → str

Retrieves the name of the most recent kernel at NAIF.

Parameters

- **naif_base_url** (*str*) – URL to search for filenames matching kernel_file_regex
- **kernel_file_regex** (*str*) – Regular expression to match filenames on the naif website
- **allowed_attempts** (*int, Optional*) – Number of allowed download times for naif page default = 3

Returns
Returns the file name of the latest kernel on the naif page (e.g., "naif0012.tls")

Return type
str

libera_utils.spice_utils.**ls_kernel_coverage**(*kernel_type: str*, *verbose: bool = False*) → dict

List time coverage of all furnished kernels of a given type

Parameters

- **kernel_type** (*str*) – Either 'CK' or 'SPK'
- **verbose** (*bool*) – If True, print to stdout also

Returns
Key is filename, value is a list of tuples giving the start and end times in ET.

Return type
dict

libera_utils.spice_utils.**ls_kernels**(*verbose: bool = False*, *log: bool = False*) → list

List all furnished spice kernels.

Parameters

- **verbose** (*bool*) – If True, print to stdout also
- **log** (*bool*) – Whether or not to log the current kernel pool (this gets called a lot)

Returns
A list of KernelFileRecord named tuples.

Return type
list

libera_utils.spice_utils.**ls_spice_constants**(*verbose: bool = False*) → dict

List all constants in the Spice constant pool

Parameters
verbose – If true, print to stdout also

Returns
Dictionary of kernel constants

Return type
dict

3.1.12 libera_utils.time

Module for dealing with time and time conventions

Some convention for this module

1. Only decorate direct spiceypy wrapper functions with the ensure_spice decorator. They should directly call a spiceypy function.

2. All spiceypy wrapper functions should read as <spiceypyfunc>_wrapper. We really only use these to allow array inputs for spiceypy functions that aren't already vectorized in C and to wrap them in ensure_spice.

3. All functions should have robust type-hinting.

Functions

convert_cds_integer_to_datetime(satellite_time)	Helper function to convert a satellite time given as an CCSDS Day Segmented Time Code (CDS) form as 8 byte integer to a timezone aware datetime object
et2utc_wrapper(et, fmt, prec)	Convert ephemeris times to UTC ISO strings.
et_2_datetime(et)	Convert ephemeris time to a python datetime object by first converting it to a UTC timestamp.
et_2_timestamp(et[, fmt])	Convert ephemeris time to a custom formatted timestamp (default is lowercase version of ISO).
sce2s_wrapper(et)	Convert ephemeris times to SCLK string https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/sce2s_c.html Decorated wrapper for spiceypy.sce2s that will automatically furnish the latest metakernel and retry if the first call raises an exception.
scs2e_wrapper(sclk_str)	Convert SCLK strings to ephemeris time.
utc2et_wrapper(iso_str)	Convert UTC ISO strings to ephemeris times.

libera_utils.time.convert_cds_integer_to_datetime

libera_utils.time.**convert_cds_integer_to_datetime**(*satellite_time: int*)

Helper function to convert a satellite time given as an CCSDS Day Segmented Time Code (CDS) form as 8 byte integer to a timezone aware datetime object

Parameters
satellite_time (*int*) – A 64-bit unsigned integer that represents CDS time

Returns
cds_time

Return type
datetime.datetime

libera_utils.time.et2utc_wrapper

libera_utils.time.**et2utc_wrapper**(*et: float | Collection[float] | ndarray*, *fmt: str*, *prec: int*) → str | Collection[str]

Convert ephemeris times to UTC ISO strings. https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/et2utc_c.html Decorated wrapper for spiceypy.et2utc that will automatically furnish the latest metakernel and retry if the first call raises an exception.

Parameters

- **et** (*Union[float, Collection[float], numpy.ndarray]*) – The ephemeris time value to be converted to UTC.
- **fmt** (*str*) – Format string defines the format of the output time string. See CSPICE docs.
- **prec** (*int*) – Number of digits of precision for fractional seconds.

Returns
UTC time string(s)

Return type
Union[numpy.ndarray, str]

libera_utils.time.et_2_datetime

libera_utils.time.**et_2_datetime**(*et: float | Collection[float] | ndarray*) → datetime | ndarray

Convert ephemeris time to a python datetime object by first converting it to a UTC timestamp.

Parameters
et (*float or Collection or numpy.ndarray*) – Ephemeris times to be converted.

Returns
Object representation of ephemeris times.

Return type
datetime.datetime or numpy.ndarray

libera_utils.time.et_2_timestamp

libera_utils.time.**et_2_timestamp**(*et: float | Collection[float] | ndarray*, *fmt: str = '%Y%m%dT%H%M%S.%f'*) → str | Collection[str]

Convert ephemeris time to a custom formatted timestamp (default is lowercase version of ISO).

Parameters

- **et** (*Union[float, Collection[float], numpy.ndarray]*) – Ephemeris Time to be converted.
- **fmt** (*str, Optional*) – Format string as defined by the datetime.strftime() function.

Returns
Formatted timestamps

Return type
Union[str, Collection[str]]

libera_utils.time.sce2s_wrapper

libera_utils.time.**sce2s_wrapper**(*et: float | Collection[float] | ndarray*) → str | ndarray

Convert ephemeris times to SCLK string https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/sce2s_c.html Decorated wrapper for spiceypy.sce2s that will automatically furnish the latest metakernel and retry if the first call raises an exception.

Parameters
et (*Union[float, Collection[float], numpy.ndarray]*) – Ephemeris time

Returns
SCLK string

Return type
Union[str, Collection[str]]

libera_utils.time.scs2e_wrapper

libera_utils.time.**scs2e_wrapper**(*sclk_str: str | Collection[str]*) → float | ndarray

Convert SCLK strings to ephemeris time. https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/scs2e_c.html Decorated wrapper for spiceypy.scs2e that will automatically furnish the latest metakernel and retry if the first call raises an exception.

Parameters
sclk_str (*Union[str, Collection[str]]*) – Spacecraft clock string

Returns
Ephemeris time

Return type
Union[float, numpy.ndarray]

libera_utils.time.utc2et_wrapper

libera_utils.time.**utc2et_wrapper**(*iso_str: str | Collection[str]*) → float | ndarray

Convert UTC ISO strings to ephemeris times. https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/utc2et_c.html Decorated wrapper for spiceypy.utc2et that will automatically furnish the latest metakernel and retry if the first call raises an exception.

Parameters
iso_str (*Union[str, Collection[str]]*) – The UTC to convert to ephemeris time

Returns
Ephemeris time

Return type
float or numpy.ndarray

libera_utils.time.**convert_cds_integer_to_datetime**(*satellite_time: int*)

Helper function to convert a satellite time given as an CCSDS Day Segmented Time Code (CDS) form as 8 byte integer to a timezone aware datetime object

Parameters
satellite_time (*int*) – A 64-bit unsigned integer that represents CDS time

Returns
cds_time

Return type
datetime.datetime

libera_utils.time.**et2utc_wrapper**(*et: float | Collection[float] | ndarray*, *fmt: str*, *prec: int*) → str | Collection[str]

Convert ephemeris times to UTC ISO strings. https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/et2utc_c.html Decorated wrapper for spiceypy.et2utc that will automatically furnish the latest metakernel and retry if the first call raises an exception.

Parameters

- **et** (*Union[float, Collection[float], numpy.ndarray]*) – The ephemeris time value to be converted to UTC.
- **fmt** (*str*) – Format string defines the format of the output time string. See CSPICE docs.
- **prec** (*int*) – Number of digits of precision for fractional seconds.

Returns
UTC time string(s)

Return type
Union[numpy.ndarray, str]

libera_utils.time.**et_2_datetime**(*et: float | Collection[float] | ndarray*) → datetime | ndarray

Convert ephemeris time to a python datetime object by first converting it to a UTC timestamp.

Parameters
et (*float or Collection or numpy.ndarray*) – Ephemeris times to be converted.

Returns
Object representation of ephemeris times.

Return type
datetime.datetime or numpy.ndarray

libera_utils.time.**et_2_timestamp**(*et: float | Collection[float] | ndarray*, *fmt: str = '%Y%m%dT%H%M%S.%f'*) → str | Collection[str]

Convert ephemeris time to a custom formatted timestamp (default is lowercase version of ISO).

Parameters

- **et** (*Union[float, Collection[float], numpy.ndarray]*) – Ephemeris Time to be converted.
- **fmt** (*str, Optional*) – Format string as defined by the datetime.strftime() function.

Returns
Formatted timestamps

Return type
Union[str, Collection[str]]

libera_utils.time.**sce2s_wrapper**(*et: float | Collection[float] | ndarray*) → str | ndarray

Convert ephemeris times to SCLK string https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/sce2s_c.html Decorated wrapper for spiceypy.sce2s that will automatically furnish the latest metakernel and retry if the first call raises an exception.

Parameters
et (*Union[float, Collection[float], numpy.ndarray]*) – Ephemeris time

Returns
SCLK string

Return type
Union[str, Collection[str]]

libera_utils.time.**scs2e_wrapper**(*sclk_str: str | Collection[str]*) → float | ndarray

Convert SCLK strings to ephemeris time. https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/scs2e_c.html Decorated wrapper for spiceypy.scs2e that will automatically furnish the latest metakernel and retry if the first call raises an exception.

Parameters
sclk_str (*Union[str, Collection[str]]*) – Spacecraft clock string

Returns
Ephemeris time

Return type
Union[float, numpy.ndarray]

libera_utils.time.**utc2et_wrapper**(*iso_str: str | Collection[str]*) → float | ndarray

Convert UTC ISO strings to ephemeris times. https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/cspice/utc2et_c.html Decorated wrapper for spiceypy.utc2et that will automatically furnish the latest metakernel and retry if the first call raises an exception.

Parameters
iso_str (*Union[str, Collection[str]]*) – The UTC to convert to ephemeris time

Returns
Ephemeris time

Return type
float or numpy.ndarray

3.1.13 libera_utils.version

Module for anything related to package versioning

Functions

version()	Get package version from metadata

libera_utils.version.version

libera_utils.version.**version**()
Get package version from metadata

libera_utils.version.**version**()
Get package version from metadata

CHAPTER

FOUR

VERSION CHANGES

4.1 2.3.1 (released)

- Fix os.path.join bug in filenaming module that broke mocked S3 paths and also fix typehinting

4.2 2.3.0 (released)

- Create CLI tools for AWS ECR image upload and Step Function triggering
- Update manifest filenames to use ULID instead of timestamp for unique identifiers
- Change logutil configure_task_logging to optionally log JSON to console
- Allow configure_task_logging to optionally propagate DEBUG messages from specific loggers
- Update documentation for how the database is used in the Libera project in DynamoDB
- Create tools for DynamoDB in AWS for .pds files (CONS and PDS)
- Replace the use of PostgreSQL with DynamoDB for the Libera project

4.3 2.2.0 (released)

- Add AnyFilename polymorphic class
- Change filename of all products to a LiberaDataProductFilename that inherits from AnyFilename
- Update filenaming convention to be all capital letters
- Improve API for manifest module
- Add prefixing to Filename classes for predictable archive paths
- Add prefixing for manifest files for predictable and navigable paths in s3 buckets
- Update git to include lfs and move test data to lfs
- Improve database manager including caching improvements
- Improve smart_copy_file and bug fixes to smart_open testing
- Refactoring and improving pds ingest for database entries and integration testing in CDK
- Added handling of construction records and pds files appropriately when ingesting
  - This includes reading a construction record and removing the pds file entry for the construction record itself

- Improved testing of pds ingest and pds file orm models to more accurately reflect use cases
- Added output manifest creation from input manifest to match timestamps in filenames of input and output manifests
- Refactored pds ingest to use AnyPath objects for handling file locations
- Added error handling to pds ingest

4.4 2.1.1 (released)

- Update dependency specification to speed up dependency resolution wrt botocore/urllib3
- Improve database initialization to work with libera_cdk changes
- Fix bug in Dockerfile that incorrectly set the default entrypoint
- Add preliminary instrument kernel

4.5 2.1.0 (released)

- Improve API to Manifest and Manifest.add_files
- Add manifest filename enforcement to Manifest class
- Update filenaming conventions for product filenames and SPICE kernels
- Allow adding an s3 bucket/prefix as a basepath for filenames

4.6 2.0.1 (released)

- Remove the extras dependency spec because of the way SQLAlchemy imports models

4.7 2.0.0 (released)

- Add filenaming classes
- Add manifest file class
- Add construction record parser
- Update DB schema to store construction records
- Update kernel generation CLI to use manifest file pattern
- Shift database and spice related libraries to extras (not installed by default)
- Add smart_copy_file function that can copy files to and from S3 and filesystem locations transparently
- Remove HDF-EOS5 filehandling code
- Add quality flag classes
- Change license to BSD3

4.8 1.0.0 (released)

- Stub out project structure
- Add build and release processes to readme
- Switch to Poetry for project dependency configuration and build management
- Add geolocation module
- Add tools in spiceutil module for caching SPICE kernels from NAIF
- Add missing unit testing coverage
- Add spice.md documentation on how the package uses and manages SPICE kernels
- Add database tooling, dev database, and ORM setup
- Add smart_open for opening local or S3 objects
- Add logging utility functions for setting up application logging

CHAPTER

FIVE

LIBERA SCIENCE DATA PROCESSING UTILITIES

Libera Utils is a package containing modules that are commonly used throughout the Libera Science Data Center codebase and processing algorithms. This package is published on PyPI to support our L2 algorithm developers with standardized code for interacting with our AWS resources and a consistent API for common tasks required of all developers.

5.1 Documentation

Documentation site, including full API listing: https://libera-utils.readthedocs.io

Additional documentation helpful for Level 2 Algorithm Developers is also available in the Libera SDC Developer Guide. Please contact the Libera SDC Team at LASP for access to the Developer Guide.

5.2 Installation

```
pip install libera-utils
```

Release candidate versions (version strings suffixed with `rc` followed by the candidate number, e.g. `1.2.3rc2`) may also be available but are likely to contain bugs.

CHAPTER

SIX

DEVELOPING LIBERA UTILS

Libera Utils is versioned formally and released as new features are made available and bugs are fixed. You can find the complete release history on PyPI. Release candidate (rc) versions are also released in order for the SDC Team to test new functionality without breaking downstream code using generous dependency specifications.

We recommend pinning major and minor release versions (e.g. 2.2) as minor releases may contain minor breaking changes. Patch releases will be restricted to bug fixes that do not cause breaking changes to existing APIs.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W