scorevideo_lib Documentation

Release 0.1.0

U8N WXD

Jun 05, 2019

Contents:

CHAPTER 1

Getting Started

Our code is hosted here: https://github.com/U8NWXD/scorevideo_lib

1.1 Getting the Code and Dependencies

1. Choose where you want to download the code, and navigate to that directory. Then download the code.

```
$ cd path/to/desired/directory
$ git clone https://github.com/U8NWXD/scorevideo_lib
```

2. Install python 3 from https://python.org or via your favorite package manager
3. Install `virtualenv`

```
$ pip3 install virtualenv
```

4. If you get a note from `pip` about `virtualenv` not being in your `PATH`, you need to perform this step. `PATH` is a variable accessible from any bash terminal you run, and it tells bash where to look for the commands you enter. It is a list of directories separated by `:`. You can see yours by running `echo $PATH`. To run `virtualenv` commands, you need to add python's packages to your `PATH` by editing or creating the file `~/.bash_profile` on MacOS. To that file add the following lines:

```
PATH="<Path from pip message>:$PATH"
export PATH
```

5. Then you can install dependencies into a virtual environment

```
$ cd scorevideo_lib
$ virtualenv -p python3 venv
$ source venv/bin/activate
$ pip install -r requirements.txt
```

Now you're ready to use the library! You can check out the API reference here.

Note: If your data is from dyad assays and structured accordingly, you can transfer lights-on marks my running the `transfer_lights_on_marks.py` tool in the directory of log files like so: `python transfer_lights_on_marks.py`. If you aren't sure if these requirements are met, they probably aren't. This is only useful for a few researchers.

CHAPTER 2

Contributing

2.1 Your First Contribution

1. Create a fork of this repository on GitHub under your own account.
2. Follow the Getting Started instructions, substituting references to the main repository for your fork.
3. Create a new branch

```
$ git checkout -b my-new-branch
```

4. Make some awesome commits

```
$ # Make some changes
$ git commit
```

5. Make sure all tests pass

```
$ ./test.sh
$ # All tests should pass, and pylint and mypy should raise no complaints
```

6. Merge in any changes from the main repository that might have occurred since you made the fork. Fix any merge conflicts

```
$ git checkout master
$ git pull upstream master
$ git checkout my-new-branch
$ git merge master
```

7. Push the branch:

```
$ git push -u origin my-new-branch
```

8. Submit a pull request on GitHub

9. Thanks for your contribution! One of the maintainers will get back to you soon with any suggested changes or feedback.

2.2 Guidelines

Any code contributions should follow the following guidelines.

2.2.1 Code Style

Python code should conform to the PEP8 style guidelines.

Docstrings should conform to the Google Style. For example (copied from Google's Style Guide):

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
```

2.2.2 Testing

To run all tests, execute `test.sh`. These tests are checked are run by Travis CI on all pull requests and the master branch. Before each commit, run `test.sh` and ensure that all tests pass. All tests should pass on each commit to make reverting easy.

Unit Testing

Unit testing is performed using pytest. To run these tests, execute `python -m pytest` from the repository root.

Code and Style Analysis

PEP8 are checked by `pylint`. `pylint` also performs static code analysis to catch some programming errors. This analysis is intended to be a fall-back defense, as unit testing should be thorough.

Type Checking

All code should use type hints wherever type cannot be inferred. At a minimum, all function prototypes should have type hints for the return value and each parameter. Type hinting is performed in the code itself, not in docstrings. Static type analysis is performed by `mypy`

Code Coverage

When running the test suite using `test.sh`, code coverage is computed by pytest-cov when running `pytest` and output after test results. Use these results to ensure that all tests are being covered. If the total coverage is not `100%`, run `coverage report -m` to see which lines were not tested. Incomplete coverage may be acceptable if the untested lines should not have been tested (e.g. code stubs for un-implemented functions).

Coverage is tracked by Codecov, which serves the badge at the top of this README.

CHAPTER 3

scorevideo_lib

3.1 scorevideo_lib package

3.1.1 Submodules

3.1.2 scorevideo_lib.add_marks module

Add marks from annotations (behaviors) in other logs

scorevideo_lib.add_marks.**copy_mark**(*logs: List[Tuple[scorevideo_lib.parse_log.Log, datetime.timedelta, int]], src_pattern: str, dest: scorevideo_lib.parse_log.RawLog, dest_label: str*) → scorevideo_lib.parse_log.RawLog

Copy a behavior into another log file as a mark, adjusting time and frame

Time and frame are adjusted so as to be correct (potentially by being negative) in relation to the other entries in `dest`. The logs are aligned in time using the provided start time and frame information.

Parameters

- **logs** – List of tuples containing log to search for `src_pattern` in and account for when adjusting time and frame, time at which the next video (`dest` for last video) starts, and frame at which the next video (`dest` for last video) starts
- **src_pattern** – Search pattern (regular expression) that identifies the behavior to copy
- **dest** – Log to insert mark into
- **dest_label** – Label for inserted mark

Returns A copy of `dest`, but with the new mark inserted

scorevideo_lib.add_marks.**copy_mark_disjoint**(*logs: List[scorevideo_lib.parse_log.Log], src_pattern: str, dest: scorevideo_lib.parse_log.RawLog, dest_label: str*) → scorevideo_lib.parse_log.RawLog

Copy a behavior into another log file as a mark, adjusting time and frame

Time and frame are adjusted so as to be correct (potentially by being negative) in relation to the other entries in `dest`, assuming that the logs in `logs` are in order, consecutive, and non-overlapping and that `dest` begins immediately after the last behavior scored in the last log of `logs`.

Parameters

- **logs** – List of consecutive and non-overlapping logs to search for `src_pattern` in and account for when adjusting time and frame
- **src_pattern** – Search pattern (regular expression) that identifies the behavior to copy
- **dest** – Log to insert mark into
- **dest_label** – Label for inserted mark

Returns A copy of `dest`, but with the new mark inserted

scorevideo_lib.add_marks.**get_ending_behav**(*behavs: List[scorevideo_lib.parse_log.BehaviorFull], end_descriptions: List[str]*) → scorevideo_lib.parse_log.BehaviorFull

Get the behavior whose description is found in a list

Parameters

- **behavs** – List of behaviors whose descriptions to search through
- **end_descriptions** – List of descriptions to search for

Returns: The first behavior whose description is found in the list

scorevideo_lib.add_marks.**get_ending_mark**(*marks: List[scorevideo_lib.parse_log.Mark]*) → scorevideo_lib.parse_log.Mark

Get the mark that has `END_MARK` as its `Mark.name`

Parameters **marks** – List of marks to search through

Returns The identified `Mark`

Raises `ValueError` – When no matching mark is found

3.1.3 scorevideo_lib.base_utils module

Basic utilities for generally applicable functions

class scorevideo_lib.base_utils.**BaseOps**

Bases: `object`

Superclass for basic operations

scorevideo_lib.base_utils.**add_to_partition**(*elem: str, partitions: List[List[str]], is_equiv: Callable[[str, str], bool]*)

Helper function to add an element to an appropriate equivalence class

Adds the element to an existing class if one is available or creates a new class by adding a partition if necessary.

Parameters

- **elem** – The element to add
- **partitions** – The list of equivalence classes to add `elem` to
- **is_equiv** – A function that accepts two elements of lst and returns whether those elements should be in the same equivalence class. For proper functioning, should implement an equivalence relation.

Returns: The equivalence classes provided but with `elem` added.

scorevideo_lib.base_utils.**equiv_partition**(*lst: Iterable[str], is_equiv: Callable[[str, str], bool]*) → List[List[str]]

Splits elements into equivalence classes using a provided callback

Parameters

- **lst** – The elements to divide in to equivalence classes. Is not modified.
- **is_equiv** – A function that accepts two elements of lst and returns whether those elements should be in the same equivalence class. For proper functioning, should implement an equivalence relation.

Returns: A list of the partitions. Each element will be in exactly one partition.

scorevideo_lib.base_utils.**remove_trailing_newline**(*s: str*)

Remove a single trailing newline if it exists in a string

```
>>> remove_trailing_newline('s\n')
's'
>>> remove_trailing_newline('s')
's'
>>> remove_trailing_newline('s\n\n')
's\n'
```

Parameters **s** – The string to remove a newline from

Returns: s, but without a terminal trailing newline, if it was present

3.1.4 scorevideo_lib.exceptions module

Custom exceptions

exception scorevideo_lib.exceptions.**FileFormatError**

Bases: Exception

Raised when a file is improperly formatted.

The message should describe the file and how it is mis-formatted.

static from_lines(*filename*, *found_line*, *expected_line*)

Create new object with message from parameters.

Parameters

- **filename** – Name of file that is improperly formatted
- **found_line** – The line that was found in the file
- **expected_line** – The line that was expected to be found

Returns: None

3.1.5 scorevideo_lib.parse_log module

Parse log files

class scorevideo_lib.parse_log.**BehaviorFull**(*behavior_line: str*)

Bases: *scorevideo_lib.parse_log.SectionItem*

Store an interpreted representation of a behavior from the full section

frame
A positive integer representing the frame number on which the behavior was scored.

time
A :py:class:timedelta object that represents the time elapsed from the start of the clip to the behavior being scored. This is a representation of the time listed in the log line.

description
The name of the behavior that appears as the second-to-last element in the provided line

subject
Always the string `either`

startend
Optional, either `start` or `end`

static validate_subject(*subject: str*) → bool
Check whether `subject` is a valid subject element

To be valid, `subject` must be exactly `either`

```
>>> BehaviorFull.validate_subject("either")
True
>>> BehaviorFull.validate_subject(" either")
False
```

Parameters **subject** – Potential subject element of a log to check

Returns: `True` if `subject` is valid, `False` otherwise

class scorevideo_lib.parse_log.**Log**
Bases: *scorevideo_lib.base_utils.BaseOps*

Store a parsed version of a log file

This version stores only the information contained in the log, not any information tied to a particular file (e.g. file name, reference to file, number of spaces separating columns).

full
A list of *BehaviorFull* objects, each representing a line from the log file's `FULL` section

marks
A list of *Mark* objects, each representing a mark from the log file

extend(*log: scorevideo_lib.parse_log.Log*) → None
Add each element of each section of a log to the current log.

Parameters **log** – Log to add elements from

Returns None

classmethod from_file(*log_file*) → scorevideo_lib.parse_log.Log
Create a *Log* object from a file

Parameters **log_file** – File to read from

Returns A parsed representation of `log_file`

classmethod from_log(*log: scorevideo_lib.parse_log.Log*) → scorevideo_lib.parse_log.Log
Create a *Log* object from another *Log* object

Parameters **log** – The object to copy

Returns A copy of the `log` parameter

classmethod from_raw_log(*log: scorevideo_lib.parse_log.RawLog*) → scorevideo_lib.parse_log.Log

Create a `Log` from a *RawLog* object

In the process, the log lines are parsed into their respective objects. This process is lossy.

Parameters **log** – The object to parse and to create the object from

Returns A parsed version of `log`

sort_lists() → None

Sort the lists of parsed material as applicable

Returns None

class scorevideo_lib.parse_log.**Mark**(*frame: int*, *time: datetime.timedelta*, *name: str*)

Bases: *scorevideo_lib.parse_log.SectionItem*

Store a `mark` from the `MARKS` section

frame

An integer representing the frame number at which the mark is placed

time

A :py:class:timedelta object that represents the time elapsed from the start of the clip to the mark. This is a representation of the time listed in the log line. Negative times are supported and are represented as their absolute times prefixed with a −.

name

Name of the mark that describes its meaning

classmethod from_line(*line: str*) → scorevideo_lib.parse_log.Mark

Create a new :py:class:Mark from a provided line from the log file

```
>>> mark = Mark.from_line("54001    30:00.03    video end")
>>> mark.frame
54001
>>> mark.time
datetime.timedelta(seconds=1800, microseconds=30000)
>>> mark.name
'video end'
```

Parameters **line** – A line from the `MARKS` section of a log file

Returns None

Raises `TypeError` – When the provided line does not conform to the expected format. Notably, all 3 elements of the line must be separated from each other by at least 2 spaces.

static time_to_str(*time: datetime.timedelta*) → str

Converts a `timedelta` object into a string

```
>>> Mark.time_to_str(timedelta(seconds=1800.07))
'30:00.07'
>>> Mark.time_to_str(timedelta(seconds=4.4557))
'0:04.45'
>>> Mark.time_to_str(timedelta(seconds=3600.5))
'1:00:00.50'
>>> Mark.time_to_str(timedelta(seconds=-1800.07))
'-30:00.07'
```

Parameters **time** – The time to turn into a string.

Returns A string representation of the time, with 2 decimal-places of second precision. The result is truncated if necessary.

Raises ValueError – Raised if time is greater than 1 day.

to_line(*other_line: str*) → str

Converts a :py:class:Mark object into a log line in the MARKS section

other_line is used as a template. It should come from the log file the returned line will be inserted into. Only loose error checking is performed, and invalid lines may produce undefined output. Similarly, if the constructed line cannot fit into the format prescribed by other_line, the output is undefined.

```
>>> mark = Mark(734, timedelta(seconds=1800.07), "video end")
>>> mark.to_line("  1    0:00.03    video start")
'734    30:00.07    video end'
```

Parameters **other_line** – A line from the MARKS section into which the resulting string could be inserted. This defines the format this method will attempt to match.

Returns: A log line that could be inserted into the MARKS section of the log from which other_line came.

Raises ValueError – Raised if other_line is invalid or the mark's time is greater than 1 day

to_line_tab() → str

Converts a :py:class:Mark object into a log line in the MARKS section

The resulting line is delimited by 4 spaces.

```
>>> mark = Mark(734, timedelta(seconds=1800.07), "video end")
>>> mark.to_line_tab()
'734    30:00.07    video end'
```

Returns A log line that could be inserted into the MARKS section of the log from which other_line came. Note that since the line has a fixed delimiter, this line may not appear to match the columns in the file. However, this delimitation is assumed by some other programs for scorevideo logs, including behaviorcode.

Raises ValueError – Raised if other_line is invalid or the mark's time is greater than 1 day

class scorevideo_lib.parse_log.**RawLog**

Bases: *scorevideo_lib.base_utils.BaseOps*

Store an interpreted form of a log file and perform operations on it

header

List of the lines in the header section

video_info

List of the lines in the video info section

commands

List of the lines in the commands section

raw
List of the lines in the raw log section

full
List of the lines in the full log section

notes
List of the lines in the notes section

marks
List of the lines in the marks section

classmethod from_file(*log_file*) → scorevideo_lib.parse_log.RawLog
Parse log file into its sections.

Populate the attributes of the RawLog class by using the get_section_* static methods to extract sections that are stored in attributes.

Parameters **log_file** – An open file object that points to the log file to read.

classmethod from_raw_log(*raw_log: scorevideo_lib.parse_log.RawLog*) → scorevideo_lib.parse_log.RawLog
Make a copy of a *RawLog* object by copying each attribute

Parameters **raw_log** – Object to copy

Returns Copy of `raw_log`

static get_section(*log_file, start: str, header: List[str], end: str*) → List[str]
Get an arbitrary section from a log file.

Extract an arbitrary section from a log file. The section is defined by a line at its start and a line at its end, neither of which are considered part of the section (not returned). A header section is also specified, the lines of which will be checked and excluded from the section. A header starts on the line immediately following the start line. If the header is not found, or if a line in it does not match, a FileFormatError is raised. If the end of the file is unexpectedly found before completing a section, a FileFormatError is raised.

Parameters

- **log_file** – An open file object that points to the log file to read. The file object must be ready to be read, and it should be at the start of the file.
- **start** – Line that signals the start of the section
- **header** – List of lines that form a header to the section. If no header should be present, pass an empty list.
- **end** – Line that signals the end of the section

Returns A list of the lines making up the section in sequential order, with each line a separate element in the list. Newlines or return carriages are stripped from the ends of lines.

static get_section_commands(*log_file*) → List[str]
Get the commands section of a log.

Extract the commands section (headed by the line "COMMAND SET AND SETTINGS") used in generating the log file. This section specifies the key commands (letters) used to signal the beginning and end of each behavior.

Parameters **log_file** – An open file object that points to the log file to read. The file object must be ready to be read, and it should be at the start of the file.

Returns A list of the lines making up the section in sequential order, with each line a separate element in the list. Newlines or return carriages are stripped from the ends of lines.

static get_section_full(*log_file*) → List[str]
Get the full log section of a log.

Extract the section of the log that contains the full scoring log. This section contains the frame number and time of each scored behavior along with the full name assigned to that behavior in the commands section

Parameters **log_file** – An open file object that points to the log file to read. The file object must be ready to be read, and it should be at the start of the file.

Returns A list of the lines making up the section in sequential order, with each line a separate element in the list. Newlines or return carriages are stripped from the ends of lines.

static get_section_header(*log_file*) → List[str]
Get the header section of a log.

Extract the top section (top two lines) of a log. This section includes a statement that the log was created by scorevideo and the name of the log file.

Parameters

- **log_file** – An open file object that points to the log file to read.
- **file object must be ready to be read,** (*The*) – and it should be at the start of the file.

Returns A list of the lines making up the header in sequential order, with each line a separate element in the list. Newlines or return carriages are stripped from the ends of lines.

static get_section_marks(*log_file*) → List[str]
Get the marks section of a log.

Extract the marks section of the log, which stores the frame number and time at which the video starts and stops. Additional marks can be added here, such as when statistical analysis should begin or when fish started behaving.

Parameters **log_file** – An open file object that points to the log file to read. The file object must be ready to be read, and it should be at the start of the file.

Returns A list of the lines making up the section in sequential order, with each line a separate element in the list. Newlines or return carriages are stripped from the ends of lines.

static get_section_notes(*log_file*) → List[str]
Get the notes section of a log.

Extract the notes section of the log, which contains arbitrary notes specified by the researcher during scoring, one per line.

Parameters **log_file** – An open file object that points to the log file to read. The file object must be ready to be read, and it should be at the start of the file.

Returns A list of the lines making up the section in sequential order, with each line a separate element in the list. Newlines or return carriages are stripped from the ends of lines.

static get_section_raw(*log_file*) → List[str]
Get the raw log section of a log.

Extract the section of the log that contains the raw scoring log. This section contains the frame number and time of each scored behavior along with the key command that was scored for that behavior

Parameters **log_file** – An open file object that points to the log file to read. The file object must be ready to be read, and it should be at the start of the file.

Returns A list of the lines making up the section in sequential order, with each line a separate element in the list. Newlines or return carriages are stripped from the ends of lines.

static get_section_video_info(*log_file*) → List[str]
Get the video info section of a log.

Extract the video info section (headed by the line "VIDEO FILE SET" of a log. This section includes information about the video including format, directory, name, start and end frames, duration, frame rate (FPS), and number of subjects

Parameters **log_file** – An open file object that points to the log file to read. The file object must be ready to be read, and it should be at the start of the file.

Returns A list of the lines making up the section in sequential order, with each line a separate element in the list. Newlines or return carriages are stripped from the ends of lines.

static section_to_strings(*start: str, header: List[str], body: List[str], end: Union[str, NoneType], trailing: List[str] = None*) → List[str]
Combine a section's components into a list of strings for writing

Parameters

- **start** – The invariant line that signals the start of the section
- **header** – Any invariant header lines that follow `start`
- **body** – The variably body of the section
- **end** – The invariant line that signals the end of the section
- **trailing** – Any lines that follow the `end` line

Returns: A list of strings suitable for writing to a file. Note that the strings do not end in a newline.

to_lines() → List[str]
Convert the current RawLog into the strings for writing to a file

Returns: A list of strings (that do not end in newlines) that can be written to a file to create a properly-formatted log file.

class scorevideo_lib.parse_log.**SectionItem**
Bases: *scorevideo_lib.base_utils.BaseOps*

Superclass for entries in a section of a log

static split_line(*line: str*) → List[str]
Split a RawLog file line in a section into its elements

Elements must be separated by at least two spaces

```
>>> SectionItem.split_line("  hi  4  test   >?why    my4 j   ")
['hi', '4', 'test', '>?why', 'my4 j']
```

Parameters **line** – Line to split

Returns: A list of the elements in the provided line

static str_to_timedelta(*time_str: str*) → datetime.timedelta
Convert a string representation of a time into a :py:class:timedelta

```
>>> SectionItem.str_to_timedelta("30:00.03")
datetime.timedelta(seconds=1800, microseconds=30000)
```

Parameters **time_str** – String representation of the time or duration

Returns: :py:class:timedelta object that represents the same duration or time as `time_str` does.

static validate_description(*desc: str*) → bool
Check whether `desc` is a valid behavior description

To be valid, `desc` must be made exclusively of digits, letters, commas, and spaces.

```
>>> SectionItem.validate_description("Some Description 3!")
False
>>> SectionItem.validate_description("Some Description, 3")
True
>>> SectionItem.validate_description("Some Description 3")
True
>>> SectionItem.validate_description("Some Description 3 here")
True
>>> SectionItem.validate_description("Some \n Description 3!")
False
```

Parameters **desc** – The potential behavior description to check

Returns: `True` if `desc` is valid, `False` otherwise

static validate_frame(*frame: str*) → bool
Check whether `frame` represents a valid frame number

A valid frame number is any integer. Specifically, any `frame` that is composed solely of one or more digits 0-9 is accepted. Negative frames are allowed and denoted by a prefix of −.

```
>>> SectionItem.validate_frame("-5")
True
>>> SectionItem.validate_frame("05")
True
>>> SectionItem.validate_frame("hi5")
False
>>> SectionItem.validate_frame("50")
True
>>> SectionItem.validate_frame(" 50 ")
False
```

Parameters **frame** – Potential frame number to validate

Returns: `True` if `frame` is a valid frame number, `False` otherwise

static validate_time(*time_str: str*) → bool
Check whether `time_str` represents a valid log time stamp

The following formats are accepted where # represents a digit 0-9 * `#:##.##` * `##:##.##` * `#:##:##.##` * `##:##:##.##`

A prefix of − is also allowed.

TODO: Check whether the minute and hour values are valid (i.e. <60)

Parameters **time_str** – The potential time representation to validate

Returns: `True` if `time_str` is a valid time, `False` otherwise

3.1.6 scorevideo_lib.transfer_lights_on_marks module

A tool that adds marks to scored log files based on a `LIGHTS ON` behavior

The marks are added with negative time and frame so as to accurately record when, relative to the start of the scored log file, the lights were recorded coming on.

When called directly, this script assumes that the log files are present in the current directory (`.`). Files are partitioned such that each partition holds the logs for one fish on one day. Afternoon files are ignored, and the `LIGHTS_ON` behavior in the `_1` or `_2` logs is transferred to the `_Morning` log.

WARNING: This script is NOT general. It is specific to one particular experiment. It may, however, be a useful example for other researchers.

class scorevideo_lib.transfer_lights_on_marks.**ExpectedFile**(*present: List[str] = None, absent: List[str] = None, regex: str = None*)

Bases: `object`

Describes the characteristics of a file name for matching

This is used in `PART_REQUIRED` and `PART_OPTIONAL` to describe required and allowed files.

match(*to_test: str*) → bool

Checks whether a file name matches this description.

A file matches if it satisfies every specified instance field. For example: >>> ExpectedFile([‘a’, ‘b’], [‘c’]).match(‘ab’) True >>> ExpectedFile([‘a’, ‘b’], [‘c’]).match(‘abc’) False >>> ExpectedFile([‘a’, ‘b’], [‘c’]).match(‘ac’) False >>> ExpectedFile([‘a’, ‘b’], [‘c’]).match(‘a’) False >>> ExpectedFile([‘a’, ‘b’], [‘c’], r’[abc]*.txt’).match(‘ab’) False >>> ExpectedFile([‘a’, ‘b’], [‘c’], r’[abc]*.txt’).match(‘ab.txt’) True >>> ExpectedFile([‘a’, ‘b’], [‘c’], r’[abc]*.txt’).match(‘abc.txt’) False

Parameters **to_test** – The string to check for matching

Returns `True` if and only if the file name matches.

scorevideo_lib.transfer_lights_on_marks.**batch_mark_lights_on**(*path_to_log_dir: str*) → None

Transfer `LIGHTS ON` marks en masse for all logs in a directory

The logs are partitioned using *`same_fish_and_day()`* into groups of logs that pertain to the same fish on the same day. A `LIGHTS ON` behavior in one of the aggression logs is transferred to the full scoring log, accounting for the change in reference point for frame numbers and times. The `LIGHTS ON` behavior can instead be specified in a separate lights-on log (see *`is_lights_on()`*). This log should have the same name as the log in which the `LIGHTS ON` behavior would otherwise be (before being transferred), except its name (before the terminal extension like `.txt`) should end in `_LIGHTSON` and the initials of the scorer may differ.

Parameters **path_to_log_dir** – Path to the directory of logs to process

Returns None

scorevideo_lib.transfer_lights_on_marks.**copy_lights_on**(*aggr_logs: List[scorevideo_lib.parse_log.Log], scored_log: scorevideo_lib.parse_log.RawLog, aggr_behav_des=typing.List[str]*) → scorevideo_lib.parse_log.RawLog

Copy a `LIGHTS ON` mark from aggression logs to the scored log

Parameters

- **aggr_logs** – Aggression logs are the _1 or _2 logs in which the researcher is looking for the first aggressive or submissive behavior by the focal male to begin scoring.
- **scored_log** – The scored log is the log from the video that was fully scored for behaviors.
- **aggr_behav_des** – List of behavior description sections that indicate that a particular behavior is considered aggressive or submissive for the purposes of beginning to fully score the video.

Returns: A copy of `scored_log`, but with the `LIGHTS ON` mark inserted.

scorevideo_lib.transfer_lights_on_marks.**find_scored_lights**(*partition: List[str]*) → Tuple[str, Union[str, NoneType]]

Find the full scoring and lights-on log of a partition

Full scoring logs are identified by *is_scored()*, and lights-on logs are identified by *is_lights_on()*.

Parameters **partition** – The list of file names from which to identify lights-on and full scoring logs.

Returns Tuple of file names of full scoring log and lights-on log. If no lights on log is found, `None` is returned instead.

Raises `ValueError` – If duplicate full scoring logs or lights-on logs are found, if no full scoring log is found, or if the scoring log is the same as the lights-on log.

scorevideo_lib.transfer_lights_on_marks.**get_last_name_elem**(*filename: str*) → str

Get the last underscore-delimited element of the name minus extensions

The last element is the part that distinguishes videos of the same fish on the same day. For example:

```
>>> get_last_name_elem("log050118_OB5B030618_TA23_Dyad_Morning.avi_CS")
'Morning'
>>> get_last_name_elem("log050118_OB5B030618_TA23_Dyad_2.avi_CS")
'2'
```

Parameters **filename** – The name from which to get the last element

Returns: The last element of the file, which distinguishes videos of the same fish on the same day

scorevideo_lib.transfer_lights_on_marks.**get_name_core**(*filename: str*) → str

Get the core of a filename

The core is the part of the filename that precedes the identifier that separates videos of the same fish on the same day. For example:

```
>>> get_name_core("log050118_OB5B030618_TA23_Dyad_Morning.avi_CS")
'log050118_OB5B030618_TA23_Dyad'
>>> get_name_core("log050118_OB5B030618_TA23_Dyad_1.avi_CS.txt")
'log050118_OB5B030618_TA23_Dyad'
>>> get_name_core("tmp/log050118_OB5B030618_TA23_Dyad_Morning.avi_CS")
'log050118_OB5B030618_TA23_Dyad'
```

Parameters **filename** – The filename from which to extract the core

Returns: The core of the filename

scorevideo_lib.transfer_lights_on_marks.**get_partitions**(*path_to_log_dir: str*)
Get partitioned file names from the specified directory

Files beginning with `.` are filtered out, as are any files for which *name_filter()* returns `False`. Names are partitioned using `equiv_partition()`, where equivalence is determined by *same_fish_and_day()* returning `True`. Each name includes the provided path as a prefix. Partitions are validated using *validate_partition()*.

Parameters **path_to_log_dir** – Path to the directory containing log files to partition

Returns A valid partitioning of the file names.

Raises `ValueError` – If any of the partitions fail validation

scorevideo_lib.transfer_lights_on_marks.**is_lights_on**(*filename: str*) → bool
Check whether a filename is for a lights-on log

A lights-on log has the same name as another log, but ends with `_LIGHTSON`. This signals that the `LIGHTS ON` behavior in the lights-on log should be transferred, maintaining timestamp and frame number, to the log of the same name (minus `_LIGHTSON`, and perhaps different scoring initials). Note that the terminal file extension (e.g. `.txt`) is ignored.

```
>>> is_lights_on("log050118_OB5B030618_TA23_Dyad_Morning.avi_CS.txt")
False
>>> is_lights_on("log050118_OB5B030618_TA23_Dyad_1.avi_CS_LIGHTSON.txt")
True
```

Parameters **filename** – Name of log file to check

Returns Whether the file is a lights-on log

scorevideo_lib.transfer_lights_on_marks.**is_scored**(*filename: str*) → bool
Check whether a filename is for a full scoring log

Uses *get_last_name_elem()* and checks whether the last name element is `Morning` or `Afternoon`.

```
>>> is_scored("log050118_OB5B030618_TA23_Dyad_Morning.avi_CS")
True
>>> is_scored("log050118_OB5B030618_TA23_Dyad_1.avi_CS")
False
```

Parameters **filename** – The filename to check

Returns: Whether the file is for a full scoring log

scorevideo_lib.transfer_lights_on_marks.**name_filter**(*filename: str*) → bool
Filter for filenames that should be included for processing

Includes the numbered log files, and the `Morning` log files. Excludes the `Afternoon` log files.

```
>>> name_filter("log050118_OB5B030618_TA23_Dyad_Morning.avi_CS.txt")
True
>>> name_filter("log050118_OB5B030618_TA23_Dyad_Afternoon.avi_CS.txt")
False
>>> name_filter("log050118_OB5B030618_TA23_Dyad_3.avi_CS.txt")
True
```

The `log` prefix is ignored

```
>>> name_filter("050118_OB5B030618_TA23_Dyad_Morning.avi_CS.txt")
True
>>> name_filter("050118_OB5B030618_TA23_Dyad_Afternoon.avi_CS.txt")
False
>>> name_filter("050118_OB5B030618_TA23_Dyad_3.avi_CS.txt")
True
```

Parameters **filename** – The filename to check

Returns Whether the file should be included for analysis

scorevideo_lib.transfer_lights_on_marks.**normalize_name**(*filename: str*) → str

Normalize a filename by adding a prefix `log` if not already present

```
>>> normalize_name("1.wmv_CS.txt")
'log1.wmv_CS.txt'
>>> normalize_name("log1.wmv_CS.txt")
'log1.wmv_CS.txt'
>>> normalize_name("logfoo")
'logfoo'
```

Parameters **filename** – The filename to normalize

Returns The normalized filename.

scorevideo_lib.transfer_lights_on_marks.**read_aggr_behav_list**() → List[str]

Read in the list of FM behaviors that are aggressive / submissive

Returns: List of behaviors that constitute the start of behavior, trimming off trailing whitespace

scorevideo_lib.transfer_lights_on_marks.**same_fish_and_day**(*name1: str*, *name2: str*) → bool

Check whether two files are from the same fish on the same day

Uses *get_name_core()* to see whether the names have the same core.

```
>>> same_fish_and_day("log050118_OB5B030618_TA23_Dyad_Morning.avi_CS",
↪"log050118_OB5B030618_TA23_Dyad_1.avi_CS")
True
>>> same_fish_and_day("050118_OB5B030618_TA23_Dyad_Morning.avi_CS",
↪"log050118_OB5B030618_TA23_Dyad_1.avi_CS")
True
>>> same_fish_and_day("log050118_OB5B030618_TA25_Dyad_Morning.avi_CS",
↪"log050118_OB5B030618_TA23_Dyad_1.avi_CS")
False
>>> same_fish_and_day("050118_OB5B030618_TA25_Dyad_Morning.avi_CS",
↪"log050118_OB5B030618_TA23_Dyad_1.avi_CS")
False
```

Parameters

- **name1** – One filename to check
- **name2** – One filename to check

Returns: Whether the names share a core

scorevideo_lib.transfer_lights_on_marks.**validate_partition**(*partition: List[str]*) → List[str]

Validates a partitioning of files

Ensures that no two files match an element of PART_OPTIONAL, and ensures that exactly one file matches each element of PART_REQUIRED. Also ensures that no files that don't match any element of either are present.

Parameters **partition** – The list of file names to validate

Returns A list of problem descriptions, one for each problem discovered. No problems are found if and only if [] is returned.

3.1.7 Module contents

scorevideo_lib is a library of tools that makes it easier to work with the scorevideo MATLAB program for scoring (annotating) animal behavior videos. This project is still very early in development, and many features remain to be implemented. Contributions are welcome! If you are interested in contributing, see the documentation for contributors.

If you're just looking to get started with these tools, see the getting started guide.

The code is hosted here: https://github.com/u8nwxd/scorevideo_lib

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index