aop

Release 1.0

Amélie Solveigh Hohe

Nov 02, 2023

CONTENTS

aop is a Python package that implements the aop standard v1.0 for amateur astronomical observation logs.

Note: This project is under active development.

Are you an amateur astronomer or astrophotographer who has some ambitions to document their observations in a clean, meaningful way? Maybe you are currently working on a small, home-made research project or maybe you are just struggling to remember the order of all the calibration frames you took last night. Either way, the aop standard is for you! It provides a clear and straightforward standard for the logging of amateur observations of the night sky. aop is the Python package that implements this standard. It only provides the means to do so, however, as it is meant to be implemented by a front-end application. Theoretically, you could use any app that is capable of implementing this package. We recommend the use of Amélie Hohe's Gala to improve your observation logging quality. Focus on the hobby you enjoy, and aop and Gala will do the logging for you.

On this page, you can learn everything about how to use aop. Look in the menu for an installation guide, code examples, a tutorial walking you through every function, and an extensive API reference that has you covered in any technical questions that might arise. Also be sure to check out the search function and the index function if you're searching for something specific!

aop has its documentation hosted on ReadTheDocs.

CHAPTER

ONE

INSTALLATION

To use aop, you first have to install it. Since it is a Python package, you obviously have to have Python installed. Get it from the official website if you don't have it installed already. You will also need the pip tool, but it usually ships with the Python interpreter available over the link above (if you for some reason don't have it, check with the pip documentation for help).

After you have Python and pip installed, we can shift our attention towards aop. Always install it from source using pip. First download the source code from GitHub.

The next step involves a little time spent in a command-line or terminal, so if you've never done anything like that, no worry, I'll walk you through. The process is dependent on your operating system, so please go to the specific sub-section below.

1.1 Linux or MacOS

Suppose you have stored the contents of the GitHub repository to `/home/amelie/Downloads/aop`. Now open your terminal window and navigate to the folder you stored the source code in. The terminal prompt should start out in your home directory, so if you fire it up it should look something like this:

```
amelie@ameliescomputer:~$
```

Tip: The ~ symbol refers to your home directory, so `/home/amelie/` in our example.

Now navigate to the source code directory using the following command:

```
amelie@ameliescomputer:~$ cd Downloads/aop
```

The prompt now changes to

```
amelie@ameliescomputer:~/Downloads/aop$
```

Using the `ls` command, we can inspect the contents of the directory. If you are in the correct directory, your output should look something like this:

```
amelie@ameliescomputer:~/Downloads/aop$ ls
aop  LICENSE  README.md  requirements.txt  setup.py
```

If it doesn't, search around a bit. Downloading from GitHub sometimes adds extra directories around the ones containing the actual code. You can enter those using the same `cd <name of directory>` command as above.

Tip: If you want to get out of a directory, just type `cd ..` - the two dots always refer to the parent directory of the one you're currently in, while a single dot `.` refers to the directory you're currently in.

After you have successfully found the correct source folder, whose `ls` output looks like described above, it's finally time to install the `aop` package to your system. To do so, simply type in the following command (without the dollar sign):

```
$ pip install -r requirements.txt .
```

Attention: Don't forget the extra dot after `requirements.txt`! While it seems minor, it is actually one of the most important parts of this command. As we mentioned earlier, a single dot like this refers to the current directory, so `/home/Amelie/Downloads/aop` in our case. This tells pip to interpret whatever it encounters in the current directory as the package we want to install. Without the dot it would be completely clueless!

You should see a bunch of lines thrown at you by pip, but as long as the last line says something like `Successfully installed aop-1.0`, you're golden.

Congretulations! You've now successfully installed the `aop` Python package to your computer. You can verify that it worked by trying to import it to a Python file. The most convenient way is to enter Python interactive mode by typing `python3` into your terminal prompt. You should now be seeing something like this '`>>>`' replacing your usual `amelie@ameliescomputer:~$` prompt. Now type:

```
>>> import aop
```

If there is no reaction and a new prompt (`>>>`) appears, that means it worked! You could even type `help(aop)` to receive more info about the package, etc.

1.2 Windows

The general process is pretty much the same as for UNIX-like systems (Linux and macOS), only the commands we use slightly differ. That's why I'm not going to describe the general installation process in great detail here again.

The terminal in Windows is called command line and is somewhat hidden, unfortunately. If you do not know already how to find it, enter the start menu and search for `cmd.exe`. Open that application and you are in the Windows command line.

Similarly to Linux, the command prompt will likely start in your home directory. We will again assume that you have downloaded the aop package source code from GitHub and stored it to `C:\Users\Amelie\Downloads\`. The command prompt starts like this:

```
C:\Users\Amelie>
```

You can navigate to the aop folder using the same `cd` command as on Linux.

```
C:\Users\Amelie> cd Downloads\aop

C:\Users\Amelie\Downloads\aop>
```

Now again, check that you are in fact in the correct directory! This time, however, you have to use a different command. The appropriate command for listing a folder's content on Windows is called `dir`, and it's expected output looks like this (whatever information is unnecessary is substituted for 'X'):

```
C:\Users\Amelie\Downloads\aop> dir
 Volume XXX
 Volume Serial Number is XXXX-XXXX

 Directory of C:\Users\Amelie\Downloads\aop
XX.XX.XXX  XX:XX    <DIR>          .
XX.XX.XXX  XX:XX    <DIR>          ..
XX.XX.XXX  XX:XX    <DIR>          aop
XX.XX.XXX  XX:XX             X.XXX LICENSE
XX.XX.XXX  XX:XX             X.XXX README.md
XX.XX.XXX  XX:XX                XX requirements.txt
XX.XX.XXX  XX:XX               XXX setup.py
               4 File(s),          X.XXX bytes
               3 Dir(s), XXX.XXX.XXX.XXX bytes free
```

Like previously, move around your folders until you are in the correct one, whose `dir` output looks like above (to move up, use `cd ..` again). Then execute

```
pip install -r requirements.txt .
```

Attention: Again: Mind the dot!

to install the package. You can verify it's installation by typing `python` to enter interactive mode, type

```
>>> import aop
```

and if it just prints the next '>>>', `aop` is installed on your system!

1.3 other operating systems

Unfortunately, I cannot provide you with a step-by-step tutorial here. Try searching the web for help on how to install Python packages from source in your specific OS.

CHAPTER

TWO

EXAMPLES

Soon, there will be some code examples here.

CHAPTER

THREE

TUTORIALS

Note: This page provides a step-by-step explanation of each function of aop. If you're looking for some quick references, go to the *Examples* page instead!

Think you could use aop but are not sure about exactly **how** to use it? Confused by all the super-technical API stuff? This page is here to help! In the following paragraphs, we will tackle the functionality of the aop package bit-by-bit in small, easy to understand steps. All right, let's get going!

Attention: Writing tutorials takes some time, so this page will not always be up to date. Especially when new features were added recently, there may not be a tutorial on them right away.

Note: This tutorial works with version 1.0.

3.1 Getting started

First, a few quick words on how this whole thing's gonna work. The Python programming language, that aop is written for, has a structure known as *classes*. The concept is simple: Every observing session follows some basic rules, the same parameters could be relevant and there's only so much you can do during a session.

That's why at the core of the aop package, there's a class simply called `aop.aop.Session`. Everything to do with aop requires you to work with that class (or, to be super-precise, *instances* of that class).

But before we dive too deep into the technical stuff already, let's first get a few conventions straight: Since aop is meant to be implemented front-end and hence has no real front-end interface, working with the package as is requires us to write some code. We could do this in a Python script, or in the Python interactive console. We could even use something more sophisticated like a Jupyter notebook or something. You can really use whatever you see fit, but for the purpose of this tutorial, we will assume that you use the Python console. We will therefore print '`>>>`' in front of every command, as is convention for the console. If you were using a Jupyter notebook, for example, this would be equivalent to `In[1]` and so on. Enter the console by typing `python` in a command-line interface or terminal (on UNIX-like systems, you sometimes have to type `python3` instead). See the Installation guide for help on how to get to the command- line.

All right, with all that out of the way, let's finally get our hands on some code! Of course, if you want to use aop, you first have to import it using the `import` Statement. Since the aop package contains two modules, `aop.aop` (providing the main functionality) and `aop.tools`, providing largely custom exceptions, we need to pronounce our `import` statement like so:

```
>>> from aop.aop import *
>>> from aop.tools import *
```

The asterisk symbol * indicates that we want to import all the content of those two modules. We can verify our import worked by quickly checking the current Julian Date:

```
>>> current_jd()
2460098.980502531
```

Hooray, it worked! Now that we know how to import the package into our code, we can move on to the next step.

3.2 Our first Session

CHAPTER

FOUR

API REFERENCE

This page contains auto-generated API reference documentation[1].

4.1 aop

Author
Amélie Solveigh Hohe

Contact
nina.tolfersheimer@posteo.de

License
MIT

Version
1.0

About: This package provides the background functionality for an implementation of the Astronomical Observation Protocol standard v1.0 (aop). It fully implements the standard, but is meant to be implemented by a front-end app.

Third-party dependencies are listed in requirements.txt.

4.1.1 Submodules

aop.aop

Author
Amélie Solveigh Hohe

Contact
nina.tolfersheimer@posteo.de

This module contains the main classes and functions of the aop package.

[1] Created with sphinx-autoapi

Module Contents

Classes

Session	A class representing an astronomical observing session.

Functions

current_jd(→ numpy.float64)	Returns the Julian Date for the current UTC or a custom datetime.
generate_observation_id(→ str)	This function generates a unique observation ID.
create_entry_id(→ str)	Creates a unique identifier for each and every entry in an .aop protocol.
parse_session(→ Session)	This function parses a session from memory to a new Session object.

aop.aop.**current_jd**(*time: str = 'current'*) → numpy.float64

Returns the Julian Date for the current UTC or a custom datetime.

It makes use of astropy's `Time` class to represent the datetime given as a Julian Date.

Parameters

time (`str`, optional) – An ISO 8601 conform string of the UTC datetime you want to be converted to a Julian Date. If `time` is "current", the current UTC datetime will be used, defaults to "current".

Raises

- **TypeError** – If the `time` argument is not of type `str`.
- ***InvalidTimeStringError*** – If the `time` argument is of type `str` but not interpretable as representing a time to astropy.time.Time.

Returns

The Julian Date corresponding to the datetime provided.

Return type

`numpy.float64`

aop.aop.**generate_observation_id**(*digits: int = 10*) → str

This function generates a unique observation ID.

The ID is generated as such: YYYY-mm-dd-HH-MM-SS-uuuuuuuuuu, where:

- YYYY: current UTC year
- mm: current UTC month
- dd: current UTC day
- HH: current UTC hour
- MM: current UTC minute
- SS: current UTC second
- uuuuuuuuuu: a `digits`-long unique identifier (10 digits per default)

Parameters
digits (int, optional) – The number of digits to be used for the unique identifier part of the observation ID, defaults to 10.

Returns
The generated observation ID.

Return type
str

aop.aop.**create_entry_id**(*time: str = 'current'*, *digits: int = 30*) → str

Creates a unique identifier for each and every entry in an .aop protocol. This identifier is unique even across observations.

Parameters

- **time** (str, optional) – If equal to "current", the current UTC datetime is used for entry ID creation. You can also pass an ISO 8601 conform string to time, if the time of the entry is not the current time this method is called, defaults to "current".
- **digits** (int, optional) – The number of digits to use for the unique part of the entry ID, defaults to 30.

Raises

- **TypeError** – If time is not a string.
- ***InvalidTimeStringError*** – If a string different from "current" is provided as time argument, but it is not ISO 8601 conform and therefore does not constitute a valid time string.

Returns

The entry ID generated. It follows the syntax YYYYMMDDhhmmssfffff-u, where:

- YYYY is the specified UTC year,
- MM is the specified UTC month,
- DD is the specified UTC day,
- hh is the specified UTC hour,
- mm is the specified UTC month,
- ss is the specified UTC second,
- fffff is the specified fraction of a UTC second and
- u represents 'digits' of unique identifier characters.

Return type
str

class aop.aop.**Session**(*filepath: str*, ***kwargs*)

A class representing an astronomical observing session.

The Session class provides several public methods representing different actions and events that occur throughout an astronomical observation. It is logged according to the Astronomical Observation Protocol Standard v1.0.

started = False

Whether the Session.start() method has already been called on this instance.

obsID

The unique observation ID generated using the *generate_observation_id()* function.

filepath

The path where the implementing script wants aop to store its files. This could be a part of the implementing script's installation directory, for example.

state

A status flag indicating the current status of the observing session. The class methods set this flag to either

- "running",
- "aborted" or
- "ended".

Initialized in `__init__()` to None, updated in `start()` to "running".

interrupted = False

A status flag indicating whether the session is currently interrupted. Initialized as False.

conditionDescription

A short description of the observing conditions.

temp

The temperature at the observing site in °C.

pressure

The air pressure at the observing site in hPa.

humidity

The air humidity at the observing site in %.

__repr__() → str

A Session object is represented by its attributes.

Returns
A string containing all the instance's attributes and their values in line format.

Return type
str

start(*time: str = 'current'*) → None

This method is called to start the observing session.

By not starting the observation when a Session object is created, it is possible to prepare the Session object pre-observation as well as parse existing protocols from memory into a new Session object. It changes the Session's "state" flag to "running" (attribute and parameter), as well as generating an observation ID, setting up a directory for the protocol and parameter log to live in, and writing the initial files to that directory. It also writes a Session Event "SEEV SESSION %obsID% STARTED" to .aop.

Parameters
time (str, optional) – An ISO 8601 conform string of the UTC datetime you want your observation to start. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- **PermissionError** – If the user does not have the adequate access rights for reading from or writing to the .aol or .aop file.
- *AopFileAlreadyExistsError* – If the .aop file the method tries to create already exists.
- *AolFileAlreadyExistsError* – If the .aol file the method tries to create already exists.

static __write_to_aop(*self*, *opcode: str*, *argument: str*, *time: str = 'current'*) → None

This pseudo-private method is called to update the .aop protocol file.

For the syntax, check with the Astronomical Observation Protocol Syntax Guide.

Parameters

- **opcode** (str) – The operation code of the event to be written to protocol, as described in the AOP Syntax Guide.
- **argument** (str) – Whatever is to be written to the argument position in the .aop protocol entry.
- **time** (str, optional) – An ISO 8601 conform string of the UTC datetime you want to use. Can also be "current", in which case the current UTC datetime will be used. In most cases, however, the calling method will pass its own time argument on to __write_to_aop(), defaults to "current".

Raises

PermissionError – If the user does not have the adequate access rights for writing to the .aop file.

static __write_to_aol(*self*, *parameter: str*, *assigned_value*) → None

This pseudo-private method is used to update the .aol parameter log.

It takes two arguments, the first being the parameter name being updated, the second one being the value it is assigned.

Parameters

- **parameter** (str) – The name of the parameter being updated.
- **assigned_value** (*any*) – The value the parameter should be assigned. Typically, this is a string or boolean.

Raises

- **PermissionError** – If the user does not have the adequate access rights for reading from the .aol file.
- **PermissionError** – If the user does not have the adequate access rights for writing to the .aol file.

interrupt(*time: str = 'current'*) → None

This method interrupts the session.

It sets the Session's interrupted flag to True in the instance attribute as well as the parameter attribute. After this, it writes a Session Event "SEEV SESSION INTERRUPTED" to the protocol and updates the .aol parameter log.

Parameters

time (str, optional) – An ISO 8601 conform string of the UTC datetime you want your observation to be interrupted at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- *SessionNotStartedError* – If the session has not yet been started.
- *NotInterruptableError* – If the session is not currently "running".
- *AlreadyInterruptedError* – If the session is already interrupted.

resume(*time: str = 'current'*) → None

This method resumes the session.

It sets the Session's `interrupted` flag to False in the instance attribute as well as the `parameter` attribute. After this, it writes a Session Event "SEEV SESSION RESUMED" to the protocol and updates the .aol parameter log.

Parameters

time (`str`, optional) – An ISO 8601 conform string of the UTC datetime you want your observation to be resumed at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- ***SessionNotStartedError*** – If the session has not yet been started.
- ***NotResumableError*** – If the session is not currently "running".
- ***NotInterruptedError*** – If the session is not interrupted.

abort(*reason: str*, *time: str = 'current'*) → None

This method aborts the session while providing a reason for doing so.

It sets the Session's `state` flag to "aborted" in the instance attribute as well as the `parameter` attribute. After this, it writes a Session Event "SEEV %reason%: SESSION %obsID% ABORTED" to the protocol and updates the .aol parameter log.

Parameters

- **reason** (`str`) – The reason why this session had to be aborted.
- **time** (`str`, optional) – An ISO 8601 conform string of the UTC datetime you want your observation to be aborted at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- ***SessionNotStartedError*** – If the session has not yet been started.
- ***NotAbortableError*** – If the session is not currently "running".

end(*time: str = 'current'*) → None

This method is called to end the observing session.

It sets the Session's `state` flag to "ended" in the instance attribute as well as the `parameter` attribute. After this, it writes a Session Event "SEEV SESSION %obsID% ENDED" to the protocol and updates the .aol parameter log.

Parameters

time (`str`, optional) – An ISO 8601 conform string of the UTC datetime you want your observation to be ended at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- ***SessionNotStartedError*** – If the session has not yet been started.
- ***NotEndableError*** – If the session is not currently "running".

comment(*comment: str*, *time: str = 'current'*) → None

This method adds an observer's comment to the protocol.

It writes an Observer's Comment "OBSC %comment%" to the protocol, where the AOP argument is whatever string is passed as the 'comment' argument to the method verbatim.

Parameters

- **comment** (str) – Whatever you want your comment to read in the protocol.
- **time** (str, optional) – An ISO 8601 conform string of the UTC datetime you want your comment to be added at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- *SessionNotStartedError* – If the session has not yet been started.
- *SessionStateError* – If the session is not currently "running".

issue(*severity: str*, *message: str*, *time: str = 'current'*) → None

This method is called to report an issue to the protocol.

It evaluates the `severity` argument and after that writes the corresponding ISSU (Issue) to the protocol:

- "potential"/"p": "ISSU Potential Issue: %message%"
- "normal"/"n": "ISSU Normal Issue: %message%"
- "major"/"m": "ISSU Major Issue: %message%"

Parameters

- **severity** (str) – An indicator of the issue's severity. It has to be one of the following strings:
  - "potential"
  - "p"
  - "normal"
  - "n"
  - "major"
  - "m".
- **message** (str) – A short description of the issue that is printed to the protocol verbatim.
- **time** (str, optional) – An ISO 8601 conform string of the UTC datetime you want your issue to be reported at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- *SessionNotStartedError* – If the session has not yet been started.
- *SessionStateError* – If the session is not currently "running".
- **ValueError** – If an improper value is passed in the 'severity' argument, that is anything different from:
  - "potential"
  - "p"
  - "normal"
  - "n"
  - "major"
  - "m".

point_to_name(*targets: list*, *time: str = 'current'*) → None

This method indicates the pointing to a target identified by name.

It starts by evaluating the 'targets' argument provided to the method and constructing a comma-separated list of targets, which is then written in the AOP argument position in the Pointing "POIN Pointing at target(s): %list of targets%" that is being written to the protocol.

Parameters

- **targets** (`list[any]`) – A list object that contains whatever objects represent the targets, most likely strings, but it could be any other object.
- **time** (`str`, optional) – An ISO 8601 conform string of the UTC datetime you want your pointing to be reported at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- *SessionNotStartedError* – If the session has not yet been started.
- *SessionStateError* – If the session is not currently "running".
- **TypeError** – If the `targets` argument is not of type `list`.

point_to_coords(*ra: float*, *dec: float*, *time: str = 'current'*) → None

This method indicates the pointing to ICRS coordinates.

It takes a 'ra' argument for right ascension and a 'dec' argument for declination. After that, a Pointing "Pointing at coordinates: R.A.: %ra% Dec.: %dec%" is written to the protocol. Provide decimal degrees for declination and decimal hours for right ascension.

Parameters

- **ra** (`float`) – Right ascension in the ICRS coordinate framework.
- **dec** (`float`) – Declination in the ICRS coordinate framework.
- **time** (`str`, optional) – An ISO 8601 conform string of the UTC datetime you want your pointing to be reported at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Raises

- *SessionNotStartedError* – If the session has not yet been started.
- *SessionStateError* – If the session is not currently "running".
- **TypeError** – If 'ra' or 'dec' is not of type 'float'.
- **ValueError** – If 'ra' is not 0.0h <= 'ra' < 24.0h.
- **ValueError** – If 'dec' is not -90.0° <= 'dec' <= 90.0°.

take_frame(*n: int*, *ftype: str*, *iso: int*, *expt: float*, *ap: float*, *time: str = 'current'*) → None

This method reports the taking of one or more frame(s) of the same target and the camera settings used.

It evaluates the type of frame provided in the `ftype` parameter to arrive at a string to be written to the protocol as frame type:

- "science frame"/"science"/"sf"/"s": "science"
- "dark frame"/"dark"/"df"/"d": "dark"
- "flat frame"/"flat"/"ff"/"f": "flat"
- "bias frame"/"bias"/"bf"/"b": "bias"

- “pointing frame”/”pointing”/”pf”/”p”: “pointing”

Finally, the method writes a Frame “FRAM %n% %frame type% frame(s) taken with settings: Exp.t.: %expt%s, Ap.: f/%ap%, ISO: %ISO%” to the protocol.

Parameters

- **n** (int) – Number of frames of the specified frame type and settings that were taken of the same target.
- **ftype** (str) – Type of frame, ftype must not be anything other than: * “science frame” * “science” * “sf” * “s” * “dark frame” * “dark” * “df” * “d” * “flat frame” * “flat” * “ff” * “f” * “bias frame” * “bias” * “bf” * “b” * “pointing frame” * “pointing” * “pf” * “p”.
- **iso** (int) – ISO setting that was used for the frame(s).
- **ap** (float) – The denominator of the aperture setting that was used for the frame(s). For example, if f/5.6 was used, provide ap=5.6 to the method.
- **expt** (float) – Exposure time that was used for the frame(s), given in seconds.
- **time** (str, optional) – An ISO 8601 conform string of the UTC datetime you want your frame(s) to be reported at. Can also be “current”, in which case the current UTC datetime will be used, defaults to “current”.

Raises

- *SessionNotStartedError* – If the session has not yet been started.
- *SessionStateError* – If the session is not currently “running”.
- **TypeError** – If one of the parameters is not of the required type: * n: int * ftype: str * expt: float * ap: float * iso: int
- **ValueError** – If an improper value is passed in the ‘ftype’ argument, that is anything other than:
  - ”science frame”
  - ”science”
  - ”sf”
  - ”s”
  - ”dark frame”
  - ”dark”
  - ”df”
  - ”d”
  - ”flat frame”
  - ”flat”
  - ”ff”
  - ”f”
  - ”bias frame”
  - ”bias”
  - ”bf”
  - ”b”

- ”pointing frame”
- ”pointing”
- ”pf”
- ”p”.

condition_report(*description: str = None*, *temp: float = None*, *pressure: float = None*, *humidity: float = None*, *time: str = 'current'*) → None

This method reports a condition description or measurement.

Every argument is optional, just pass the values for the arguments you want to protocol. Each argument will be processed completely separately, so a separate protocol entry will be produced for every argument you provide. For each type of condition report, a corresponding flag will be set as an instance attribute as well as a parameter. This flag update will also be written to the parameter log. In case a description is provided, a Condition Description “CDES %description%” is written to the protocol, if a temperature, pressure or humidity is provided, however, a Condition Measurement is written to protocol for each measurement provided, respectively:

- “CMES Temperature: %temp%°C” and/or
- “CMES Air Pressure: %pressure% hPa” and/or
- “CMES Air Humidity: %humidity%%”

Parameters

- **description** (str, optional) – A short description of every relevant element influencing the overall observing description, but do not provide any measurements, as these are a Condition Measurement rather than a Condition Description, defaults to None.
- **temp** (float, optional) – The measured temperature in °C, defaults to None.
- **pressure** (float, optional) – The measured air pressure in hPa, defaults to None.
- **humidity** (float, optional) – The measured air humidity in %, defaults to None.
- **time** (str, optional) – An ISO 8601 conform string of the UTC datetime you want your condition update to be reported at. Can also be “current”, in which case the current UTC datetime will be used, defaults to “current”.

Raises

- *SessionNotStartedError* – If the session has not yet been started.
- *SessionStateError* – If the session is not currently “running”.

report_variable_star_observation(*star_id: str*, *chart_id: str*, *magnitude: float*, *comparison_star_1: str*, *comparison_star_2: str = None*, *codes: list = None*, *time: str = 'current'*) → None

This method reports a (visual) observation of a variable star.

It writes the op code “VSOB” and logs several important parameters. This method is constructed with reporting your observation to the American Association of Variable Star Observers (AAVSO) in mind. Please note, however, that it DOES NOT write an AAVSO Visual File Format compliant report.

Parameters

- **star_id** (str) – An unambiguous identifier of the variable star being observed (e.g. “del Cep”).

- **chart_id** (str) – The ID of the finder chart in usage. AAVSO charts usually have a box at the upper righthand corner containing this information.
- **magnitude** (float) – Your magnitude estimate, including the decimal point.
- **comparison_star_1** (str) – The label of the first comparison star being used. AAVSO charts leave out the decimal point here, please do so as well.
- **comparison_star_2** (str, optional) – The label of the second comparison star being used, if any.
- **codes** (list, optional) – A list of comment codes detailing your observation. Usage of the official AAVSO one-character comment codes is recommended, but not mandated.
- **time** (str, optional) – An ISO 8601 conform string of the UTC datetime you want your condition update to be reported at. Can also be "current", in which case the current UTC datetime will be used, defaults to "current".

Returns
None

Raises
- *SessionNotStartedError* – If the session has not yet been started.
- *SessionStateError* – If the session is not currently "running".

aop.aop.**parse_session**(*filepath: str*, *session_id: str*) → *Session*

This function parses a session from memory to a new Session object.

Provided with the filepath to the general location where the protocol and log files are stored and an observation ID, it reads in the observation parameters from the session's parameter log. This information is then used to create a new Session object, which is returned by the function. If it has no observationID attribute, it is recreated from the function input.

Parameters
- **filepath** (*str*) – The path to the file where you expect the session directory to reside. This is most likely equivalent to the path passed to the Session class to create its files in, which in turn is most likely somewhere in the installation directory of the implementing script.
- **session_id** (*str*) – The observationID of the session to be parsed.

Raises
- *AolNotFoundError* – If there is no .aol file using the specified filepath and session_id.
- **SessionIdDoesntExistOnFilepathError** – If the specified session_id is not in the filepath provided.
- **NotADirectoryError** – If the specified filepath does not constitute a directory.

Returns
The new Session object parsed from the stored observation parameters. For all intents and purposes, this object is equivalent to the object whose parameters were used to parse, and you can use it to continue your observation session or protocol just the same. Just be careful not to run the Session.start() method again, as this would overwrite the existing protocol instead of continuing it!

Return type
Session

aop.tools

Author
Amélie Solveigh Hohe

Contact
nina.tolfersheimer@posteo.de

This module contains auxiliary classes and functions for the aop package.

Module Contents

exception aop.tools.**AolFileAlreadyExistsError**(*filepath: str*, *session_id: str*)

Bases: Exception

An error raised upon trying to initialize an .aol file that already exists.

exception aop.tools.**AolNotFoundError**(*session_id: str*)

Bases: Exception

An error raised upon trying to load an .aol file that doesn't exist.

exception aop.tools.**AopFileAlreadyExistsError**(*filepath: str*, *session_id: str*)

Bases: Exception

An error raised upon trying to initialize an .aop file that already exists.

exception aop.tools.**InvalidTimeStringError**(*invalid_string: str*)

Bases: Exception

An error raised upon providing a string to current_jd's time argument that is not interpretable as a time.

exception aop.tools.**SessionIDDoesntExistOnFilepathError**(*invalid_id: str*)

Bases: Exception

An error raised when a specified session ID could not be found on the provided filepath.

exception aop.tools.**SessionStateError**(*event: str*, *state: str*)

Bases: Exception

An error raised when the current session parameters don't allow for the requested operation.

__str__() → str

The default custom error message of SessionStateError

Returns
default custom error message

Return type
str

exception aop.tools.**NotInterruptableError**

Bases: *SessionStateError*

An error raised when trying to interrupt a session that is not currently 'running'.

Inherits from *SessionStateError*, uses "interrupt session" as impossible operation and "not running" as problematic session state.

exception aop.tools.**NotResumableError**

Bases: *SessionStateError*

An error raised when trying to resume a session that is not currently 'running'.

Inherits from *SessionStateError*, uses "resume session" as impossible operation and "not running" as problematic session state.

exception aop.tools.**NotAbortableError**

Bases: *SessionStateError*

An error raised when trying to abort a session that is not currently 'running'.

Inherits from *SessionStateError*, uses "abort session" as impossible operation and "not running" as problematic session state.

exception aop.tools.**NotEndableError**

Bases: *SessionStateError*

An error raised when trying to end a session that is not currently 'running'.

Inherits from *SessionStateError*, uses "end session" as impossible operation and "not running" as problematic session state.

exception aop.tools.**AlreadyInterruptedError**

Bases: *SessionStateError*

An error raised when trying to interrupt a session that is already 'interrupted'.

Inherits from *SessionStateError*, uses "interrupt session" as impossible operation and "interrupted" as problematic session state.

exception aop.tools.**NotInterruptedError**

Bases: *SessionStateError*

An error raised when trying to resume a session that is not currently 'interrupted'.

Inherits from *SessionStateError*, uses "resume session" as impossible operation and "not interrupted" as problematic session state.

exception aop.tools.**SessionNotStartedError**(*illegal_operation: str*)

Bases: Exception

An error raised when trying to perform a session operation before the session has been started.

__repr__() → str

Custom error message.

Returns
custom error message

Return type
str

PYTHON MODULE INDEX

a

INDEX

Symbols

A

C

E

F

G

H

I

M

N

O

P

R

S

T