aop

Release 1.1

Amélie Solveigh Hohe

Dec 29, 2023

CONTENTS

aop is a Python package 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 package is for you! It provides a clear and straightforward way for the logging of amateur observations of the night sky. However, it only provides the means to do so, 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, which will in all likelihood result in a .zip-archive called something along the lines of `aop-master` you need to unpack using your favourite zipping tool.

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-2.0`, you're golden.

Congratulations! 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 (the little Windows icon to the left of your task bar) 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 extracted 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: These tutorials work with version 1.1.

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

In this section, we will launch our first observing session using aop! We will assume that you have successfully imported the aop package as described above. If you have, creating a session is a really straightforward process. Simply type

```
>>> my_session = Session(filepath="/home/amelie/astronomy_logs/")
```

where you replace the *filepath* argument with whatever location you want your logs to be stored to.

> **Attention:** If you are on Windows and want to use the Windows-specific backslash (\) notation (e.g. C:\Users\Amelie\astronomy_logs\) you either have to put a little r in front of the first quotation mark or double each backslash. This is because Python uses the backslash as a special character and would therefore not realize this was a file path. You should, however, be able to use regular slashes for Windows paths as well.

Congratulations! You've just created your first instance of aop's *aop.aop.Session* class! That is what the `my_session` object you've just instantiated is. You can now use all the methods of the `Session` class on that object.

But before we do that, we'll dive a bit deeper into the possibilities when setting up a new session. As you've seen, aop requires you to give it a *filepath* argument to know where to store its stuff. Since this argument is required, you could technically also remove the "filepath=" part, so long as it remains the first argument.

```
>>> my_session = Session("/home/amelie/astronomy_logs/")
```

The Session constructor method, that does all the heavy lifting for us here, also excepts a wide range of other arguments, however. These are optional, so we need to state them by name. Providing information on the observer and the location would look like this, for example:

```
>>> my_session = Session(filepath="/home/amelie/astronomy_logs/",
                         observer="Jane Doe",
                         locationDescription="12 Example Road")
```

There are many more options here, check the documentation of the *aop.aop.Session* class for reference. aop does not really **do** anything with that information, other than write it to the log, so it's your call what if any you want to report, although this is mostly very basic information that shouldn't really be missing either.

The door is now wide open, but before we can do anything else, there is just one small step we need to take: We need to start the session first. It might seem counterintuitive that an aop session does not start upon creation, but this has one practical reason: Doing it like this, you can prepare your Session object in advance, and start the session whenever you're ready, which some people might find useful. Keep in mind, after all, that the aop package is really not meant

to be used in an interactive shell like we do here, but it is meant to be implemented by an app that provides a proper front-end interface and that could perhaps do something useful with that possibility.

Nonetheless, starting the session is just this simple command away:

```
>>> my_session.start()
```

And that's it! The start() method works all by itself, no arguments required. You can provide it with the `time` argument, as all `Session` methods, but that's a story for another day that is really not necessary for beginners.

aop should also now have logged it's first entry. To check it out, navigate to the file path you provided aop with when creating the my_session object in the beginning. You should see a new directory there with a somewhat cryptic name that starts with the current date in year-month-day format. This is the observation ID, that makes your specific observation unique. It consists of the date and time it was created, separated by hyphens, and then ten random characters and numbers, that provide another level of uniqueness. Move into that directory and you should see two files of the same name, but with different file extensions. There is one with extension `.aol` that we're going to ignore for now. The real stuff happens inside the `.aop` file, which you can open with any text editor (though high-level word processing applications such as LibreOffice Writer or Microsoft Word are not ideal since they would likely mess up the layout - please use something along the lines of NotePad, which should be built into all modern operating systems in some capacity, though it may be named differently).

If you go ahead and do so, you'll firstly see a bunch of meta-data that you provided above. But then, in a new paragraph, you should now see a line that starts with some gibberish in brackets, then a very large number around 2.5 million, and finally the message `SEEV SESSION observation id STARTED`. That means we were successful!

```
(20231102143452235660-73a5b82e600746e78d830499ed9ee5) 2460251.1075490238 -> SEEV SESSION␣
↪2023-11-02-14-34-52-3bb50f5e32 STARTED
```

A few more detailed notes on the contents of that line: The first part, in the brackets, is the so-called entry ID, that makes every proper entry completely unique, even across observations. It consists of the date and precise time it was created, all smashed together before the hyphen in the middle, and then 30 characters and numbers that are completely random and ensure that your entry ID is completely unique. The point of creating a log is to be able to precisely reference it in the future, after all.

The large number that follows the entry ID is the so-called *Julian Date* (JD), a system of keeping time that is often used in astronomy, since it is independent of time zones, daylight saving hours, calendar conventions, etc. It instead relies on counting the days that have passed since a largely arbitrary, yet very well defined point in the distant past. If you're curious, try to calculate which date it was (or look it up, since this stuff can get really complicated). The counting of Julian Date present here has ten decimal places, corresponding to an accuracy of a 10 billionth of a day (0.00864 milliseconds or 8.64 microseconds). That is limited by the accuracy of your device's clock, however.

After the arrow (->), that is just a visual aid to separate the technical stuff from the actual log, there is only one mystery left: What does `SEEV` mean? This is what is known to aop as an *op code*, short for operation code, and it encrypts what type of action is recorded here. Starting the session falls into the category of *"session events"* (hence the abbreviation SEEV). Everything that comes after the op code is referred to as the op code's *argument* and carries the additional information necessary for understanding what has been going on - in this case, the information that a session was started, along with it's specific observation ID (although this is technically not necessary, since the observation ID is also recorded at the top of the file with the other *session parameters* under the short handle "obsID").

You're now familiar with setting up an aop session, starting it, and you also now where to find the results and how to read them. That's a great start! In the next chapter, we will explore the other session events that are available to you.

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 characters 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 the specified amount 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.

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.

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", as well as generating an observation ID, setting up a directory for the protocol to live in, and writing the initial files to that directory.

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 .aop file.
- ***AopFileAlreadyExistsError*** – If the .aop file the method tries to create already exists.
- ***AopFileAlreadyExistsError*** – If the .aopl file the method tries to create for legacy only already exists.
- ***AolFileAlreadyExistsError*** – If the .aol file the method tries to create for legacy only already exists.

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

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

For the syntax, check with the Astronomical Observation Protocol Syntax Guide. **CAUTION!** This method should be considered deprecated and should not be used in any new code!

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 .aopl 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 .aopl file.

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

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

It takes two arguments, the first being the parameter name being updated, the second one being the value it is assigned. **CAUTION!** This method should be considered deprecated and should not be used in any new code!

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 and logs that change.

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 and logs that change.

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" and logs that change.

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" and logs that change.

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.

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.

There are three severity levels available:

- potential
- normal
- major

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 logged as well.
- **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 to 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 one or more target(s) identified by name.

It can handle multiple targets at once, each will be logged in its own sub-tag of the 'point' tag.

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.

Unlike the *point_to_name()* method, this method can only handle one set of coordinates each time, ideally representing the middle of the field of view. 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' is not of type 'float'.
- **TypeError** – If '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 same camera settings used.

It is centered on using a DSLR/DSLM as detector, since it uses the term ISO and expects aperture to be provided as a fraction, like it is common for photographic lenses. You can use a dedicated astronomy camera as well however. Interpret 'iso' as Gain and calculate the aperture fraction of your optics for the 'ap' argument. This method recognizes five distinct frame types:

- science/light frame (sometimes called 'sub', too)
- dark frame
- flat frame
- bias frame
- pointing frame

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 or Gain 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 log. Each argument will be processed completely separately, so a separate log entry will be produced for every argument you provide. For each type of condition report, a corresponding flag will be set.

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".

Returns
None

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.

Alongside your magnitude estimate, the finder chart you used as well as at least one comparison star and possible comment codes are logged. This method is very much 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, as this is a higher task left to the front-end application.

Parameters

- **star_id** (str) – An unambiguous identifier of the variable star being observed (e.g. "del Cep").
- **chart_id** – The ID of the finder chart in usage. AAVSO charts usually have a box at the upper right-hand

corner containing this information. :type chart_id: str :param magnitude: Your magnitude estimate, including the decimal point. :type magnitude: float :param comparison_star_1: The label of the first comparison star being used. AAVSO charts leave out the decimal point here, please do so as well. :type comparison_star_1: str :param comparison_star_2: The label of the second comparison star being used, if any. :type comparison_star_2: str, optional :param codes: A list of comment codes detailing your observation. Usage of the official AAVSO one-character comment codes is recommended, but not mandated. :type codes: list, optional :param time: An ISO 8601 conform string of the UTC datetime you want your

observation 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 log files are stored and an observation ID, it reads in the observation parameters from the session's log. This information is then used to create a new Session object, which is returned by the function.

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.
- **session_id** (*str*) – The observation ID of the session to be parsed.

Raises

- ***AolNotFoundError*** – If there is no .aol legacy file using the specified filepath and observation ID.
- **SessionIdDoesntExistOnFilepathError** – If the specified observation 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 log 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! Due to the 'started' flag of the new Session object most likely being set to True, however, this should generally not be possible.

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

P

R

S

T