miriad-python Documentation

Release 1.2.4

Peter Williams

June 26, 2023

Contents

Welcome to the *miriad-python* manual. If you're unfamiliar with how *miriad-python* works or what it's for, try starting at the *introduction*. For more information about the *miriad-python* project, including installation instructions, issue reporting information, releases, and academic references, please see the miriad-python website.

CHAPTER 1

Contents

1.1 An Introduction to miriad-python

miriad-python is a framework for interacting with the MIRIAD radio interferometry package via the Python programming language. It offers three main groups of tools, used for:

- *reading and using MIRIAD data files in Python*.
- *executing existing MIRIAD tasks*.
- *writing your own MIRIAD tasks in Python*.

You should get started by reading about *high-level access to datasets*.

1.2 Loading and Processing MIRIAD Data

miriad-python deals with MIRIAD data on two levels:

- There is a lightweight high-level class, `miriad.Data`, that allows you to easily store references to datasets, check whether they exist, rename them, and so on.
- There is a lower-level module, *`mirtask`*, that lets you open up datasets and access at their contents directly. Doing so successfully requires a familiarity with the details of the MIRIAD data formats, which this documentation does not attempt to provide.

If you're just getting started with *miriad-python*, it's important to understand the high-level `miriad.Data` class but not necessary to read about the low-level *`mirtask`* module just yet.

1.2.1 High-Level Access to MIRIAD Data: `miriad`

On the commandline, you refer to datasets by their filenames. The *`miriad`* module provides two fundamental classes, `VisData` and `ImData`, which analogously let you refer to datasets in a Python program. Instances of these class are lightweight and provide features to make it easy to perform common operations on your data.

To instantiate one of these classes, just call the constructor with a filename as an argument:

```
from miriad import VisData, ImData
vis = VisData ('./fx64c-3c147-1400')
im = ImData ('./residual.rm')
```

It's important to understand that these objects are *references* to datasets, and as such the underlying file doesn't have to exist when you create the object. Also, creating one of these objects is a very cheap operation.

Both `miriad.VisData` and `miriad.ImData` are subclasses of a more generic class, `miriad.Data`. Instances of this class have methods and properties that provide common functionality regarding MIRIAD datasets. One set of functionality is checking basic properties of the dataset on disk:

- `Data.exists` to see if it exists on disk.
- `Data.mtime` to check when it was last modified. This requires that the dataset exists; the variant attribute `umtime` returns unconditionally. (Hence the "u" prefix to its name.)
- `Data.realPath()` to get its canonical filename.

You can also perform some basic operations. (From here on out, we will drop the ''Data" prefix in the names we show. Also, note that you can click on the link associated with all of these function or property names to access the more detailed reference documentation for that item.)

- `moveTo()` renames a dataset.
- `copyTo()` copies it.
- `delete()` deletes it.
- `apply()` configures a MIRIAD task object (`mirexec.TaskBase`) to run on this dataset via the *`mirexec`* subsystem. See *Executing MIRIAD Tasks: mirexec* for more information. See also the verbose variant `xapply()`.

You can create more `Data` instances with filenames similar to existing ones:

- `vvis()` creates a new `VisData` instance referencing a similar filename.
- `vim()` creates a new `ImData` instance referencing a similar filename.

And you can open the dataset with `open()` to get access to its contents. See *Low-Level Access to MIRIAD Data* for more information.

You may also wish to enable tracing of MIRIAD task execution in *miriad-python* by calling `basicTrace()`. There are a few more rarely-used members of `Data` not mentioned here that are documented in the *API reference* below.

Visibility Datasets

The `VisData` subclass of `Data` has additional routines specifically useful for UV data:

- `catTo()` runs **`uvcat`** on a dataset to produce a copy of it.
- `averTo()` runs **`uvaver`** on a dataset to produce an averaged copy of it.
- `lwcpTo()` creates a "lightweight copy" of a dataset, duplicating its metadata but not the visibilities, which makes certain common operations much faster.
- `readLowlevel()` opens the dataset directly for lowlevel access to the visibility data.

Besides these routines, the `VisData` subclass implements several generic methods specified in `Data`, so you should always create a `VisData` instance when you know that you're referring to a visibility dataset.

Image Datasets

The `ImData` subclass of `Data` is used for referencing image data. It currently does not have any routines specifically applicable to image data, but it implements several of the `Data` methods correctly, so you should always create a `ImData` instance when you know that you're referring to an image dataset.

`miriad` API Reference

This section presents a detailed API reference for the *`miriad`* module.

Dataset Classes

Tracing Task Execution

The *`miriad`* module also provides infrastructure for tracing task execution and operations on datasets.

`miriad.`**`launchTrace`**
: Traces the execution of commands.

  Should be a callable or `None`. Will be called by `trace()`, which is invoked every time a MIRIAD task is executed via *`mirexec`* or a dataset is renamed, copied, or deleted. *launchTrace* should take one argument, which will be a list of strings representing the commandline that is being invoked. If none, `trace()` has no effect.

  The function `basicTrace()` sets *launchTrace* to a simple default.

1.2.2 Low-Level Access to MIRIAD Data

Is awesome. But the documentation will be written a bit later.

`mirtask` API Reference

1.2.3 MIRIAD Data Utilities: `mirtask.util`

The module *`mirtask.util`* contains miscellaneous utilities for working with MIRIAD data. It is located in the *`mirtask`* package because some of these utilities make calls into the MIRIAD subroutine library, unlike the general, pure-Python tools in the *`miriad`* module.

Antpols and Basepols

Docs here.

Document FPOL_XYRLIQUV.

Textual Conversion

Data Conversion

Utilities for Writing Tasks

Linetypes

Baselines

Polarizations

Julian Dates

Optimizers

Coordinate Manipulations

Fast-Fourier-Transform Imaging

1.3 Executing MIRIAD Tasks: `mirexec`

The *mirexec* module makes it convenient to launch tasks from within Python. The simplest invocation looks a lot like what one would run on the command line:

```
from mirexec import TaskUVFlag
TaskUVFlag (vis='fx64a-3c286-2700', select='ant(26)', flagval='f',
            noquery=True).run ()
```

If you need them, however, the *mirexec* module provides more sophisticated facilities allowing you to retrieve the output of a task, run many tasks in parallel, and so on.

Running a MIRIAD task in *mirexec* requires three steps which, as shown above, can often be condensed into a single line of Python:

1. Create an instance of a "task" class corresponding to the task you wish to run.
2. Set the task keywords and options.
3. Call a method that actually launches the task.

The following code is equivalent to the first example, but breaks down the steps more explicitly:

```
from miriad import VisData
from mirexec import TaskUVFlag
v = VisData ('fx64a-3c286-2700')
# Create instance:
t = TaskUVFlag ()
# Set some keywords:
t.vis = v
t.select = 'ant(26)'
t.flagval = 'f'
# Set some options:
t.noquery = True
t.hms = False # i.e., do *not* specify the "hms" option
# Launch task.
# Executes: uvflag vis=fx64a-3c286-2700 select=ant(26) flagval=f options=noquery
t.run ()
```

1.3.1 Creating Task Instances

The `TaskUVFlag` class shown in the above examples is a subclass of the `TaskBase` class, which provides a generic structure for invoking MIRIAD tasks. The *mirexec* module defines such subclasses for many, but far from all, of the tasks provided with MIRIAD. It's easy to create your own `TaskBase` subclass for anything you need that's not provided with *miriad-python*, however. See *below* for more information.

The `TaskBase` class provides functions for setting keyword arguments and actually invoking the task. For the full details, see the detailed API documentation. Subclasses specify the name of the particular task that is run and the keywords and options it accepts.

Task instances can be reused: you can create an object, set arguments, and run it, then change some or all of the arguments and run it again. Among other uses, this makes it easy to apply a task to several datasets:

```
t = TaskUVAver ()
t.interval = 5
t.line = 'chan,800,101'
t.nocal = True
for v in listManyDatasets ():
   # The set() method returns 'self' for easy chaining of
   # method invocations.
   t.set (vis=v, out=v.vvis ('av')).run ()
```

1.3.2 Setting Task Parameters

You can set the task parameters in several ways: as a property on the object, as in the example above, as a keyword argument to the object's constructor, or as a keyword argument to the object's `set()` method. The latter two forms are shown in the example below:

```
from miriad import VisData
from mirexec import TaskUVFlag
v = VisData ('fx64a-3c286-2700')
# This is equivalent to the previous example.
t = TaskUVFlag (vis=v, flagval='f', noquery=True)
t.select = 'ant(26)'
t.run ()
# As is this.
t.set (vis=v, select='ant(26)', flagval='f', noquery=True)
t.run ()
```

Thus, the most succinct way to execute a task is to write something like:

```
TaskUVFlag (vis=v, flagval='f', select='pol(yy)').run ()
```

The names and values of keywords in Python are mapped to command-line arguments with the following rules:

- Keyword arguments have the same name in Python as they do on the command-line if possible. If the MIRIAD keyword is a Python keyword (*e.g.*, "in"), the keyword is accessible in Python by suffixing it with an underscore ("in_").
- In most cases, the textual value of each MIRIAD keyword is the stringification of the Python variable assigned to it. If the Python value is `None`, the keyword is not supplied on the command-line.
- However, if the Python variable assigned to the keyword is a non-string iterable, the textual value of the keyword is the stringification of each item in the iterable, joined together with commas. For instance, if you run:

```
from mirexec import TaskMfCal
TaskMfCal (vis=foo, line=['chan', 60, 15]).run ()
```

the *line* keyword of `mfcal` will be `chan,60,15`.

- The keyword "options" isn't used directly. Instead, each possible option to a task is a separate field on the task object that should be set to a `bool`. The option is supplied if the field is `True`. There are rare tasks that have an option with the same name as a keyword; in those cases, the keyword is the one controlled by the property on the task object.

There are several functions that will actually execute the task. Each has different uses:

- `run()` executes the task and waits for it to finish. The task output is sent to the stdout of the Python program and the task input is set to `/dev/null`.
- `snarf()` executes a task and waits for it to finish. The task's output to its standard output and standard error streams are returned to the caller.
- `runsilent()` executes the task and waits for it to finish. The task output is sent to `/dev/null`.
- `launch()` starts the task but doesn't wait for it to finish; instead, it returns a `MiriadSubprocess` instance that allows interaction with the launched subprocess.
- `launchpipe()` starts the task but doesn't wait for it to finish. The output of the task is redirected to pipes that can be read using the `MiriadSubprocess` instance.
- `launchsilent()` starts the task but doesn't wait for it to finish. The output of the task is redirected to `/dev/null`.

1.3.3 Defining Your Own Task Classes

In most cases, it's straightforward to define your own task class. To wrap the task "newtask", you should write something like:

```
from mirexec import TaskBase

class TaskNewTask (TaskBase):
    _keywords = ['vis', 'line', 'flux', 'refant']
    _options =  ['nocal', 'nopass', 'mfs']

def demo (vis):
    t = TaskNewTask (vis=vis)
    t.flux = 1.0
    t.nocal = True
    t.run ()
```

The name of the task executable is inferred from the class name by stripping off the prefix "Task" and lower-casing the rest of the letters. If this heuristic won't work, you can specify the task name explicitly by setting *_name* on the class:

```
from mirexec import TaskBase

class DifferentNames (TaskBase):
    _name = 'newtask'
    _keywords = ['vis', 'line', 'flux', 'refant']
    _options =  ['nocal', 'nopass', 'mfs']
```

If you're feeling fancy, here's a less typing-intensive way of generating arrays of short strings:

```
from mirexec import TaskBase

class TaskNewTask (TaskBase):
    _keywords = 'vis line flux refant'.split ()
    _options =  'nocal nopass mfs'.split ()
```

1.3.4 `mirexec` API Reference

This section presents a detailed API reference for the *`mirexec`* module.

Generic Task Class

exception mirexec.**TaskFailError**(*returncode*, *cmd*)

Signals that a task exited indicating failure, though it was able to be launched.

`TaskFailError` may be a subclass of `subprocess.CalledProcessError`, if such a class exists. (It was introduced in Python 2.5.) Otherwise, it is a functional equivalent to that class.

Instances have an attribute **returncode** indicating the exit code of the task. This will be nonzero, since zero indicates success. As far as I know, all MIRIAD tasks exit with a code of 1 unless they die due to a POSIX signal (in which case, the exit code is conventionally the negative of the signal number).

Instances also have an attribute **cmd** which is a string version of the command line that was executed. The arguments are joined together with spaces, so there's potential for ambiguity if some of the argument values contain spaces.

Specific Task Classes

We try to keep this list up-to-date, but it may not be complete. If you discover a wrapped task that isn't documented here, please notify the author. As mentioned above, it's straightforward to wrap a new task yourself: see *Defining Your Own Task Classes*.

Setting up Subprocess Environment

Utility Classes

1.4 Writing Your Own MIRIAD Tasks

There are several things that you need to do to write your own MIRIAD tasks that blend in with the standard MIRIAD tasks.

1.4.1 MIRIAD-Style Argument Handling: `mirtask.keys`

Like other UNIX programs, MIRIAD tasks accept input parameters from the user via command-line arguments. The way in which MIRIAD tasks do this, however, is different than the usual UNIX way.

Each task defines a number of "keywords". The MIRIAD keyword-handling subsystem can be used to obtain zero or more values of each keyword from the command-line arguments. The values of a given keyword do not necessarily all have to be of the same type. The user specifies the values of these keywords on the command-line with an assignment syntax:

```
invert vis=3c286.uv map=3c286.mp imsize=824,724 slop=0.5 select='ant(1,3),pol(xx)'
```

Here, the *vis* keyword has a single string value (interpretable specifically as a filename), the *imsize* keyword has two integer values, and the *select* keyword has two string values. (Note that the keyword-handling routines process parentheses in string-valued keywords and do *not* consider the keyword to have the values "ant(1", "3)", and "pol(xx)".)

Boolean-valued keywords are called "options" and are implemented by an *options* pseudo-keyword:

```
invert options=mfs,double
```

Here the options *mfs* and *double* have `True` values while all other options are `False`.

Handling keywords in miriad-python

Tasks in MIRIAD obtain the values of keywords in a procedural way. In miriad-python, keywords are generally specified declaratively and their values are obtained automatically, although there is support for the more general procedural approach.

To parse arguments in a miriad-python task:

1. Instantiate a `KeySpec` object.
2. Specify the keywords your task accepts.
3. Use the `KeySpec.process()` method to obtain a data structure populated with the settings for all keywords based on your specification.

Here's a simple example:

```
from mirtask import keys

ks = keys.KeySpec ()
ks.keyword ('param', 'd', 0.25)
ks.keyword ('mode', 'a', 'deconvolve')
ks.option ('verbose', 'noop')

opts = ks.process ()
if opts.param < 0:
   die ('"param" must be positive, not %f' % opts.param)
if opts.mode not in ('deconvolve', 'stirfry'):
   die ('Unknown operation mode "%s"' % opts.mode)
if opts.verbose:
   print "Being verbose starting now!"
```

The methods on `KeySpec` to define keywords are:

- `keyword()` defines a keyword that takes a single, typed value, with a default if the keyword is left unspecified.
- `mkeyword()` defines a keyword that takes multiple values of the same type.
- `keymatch()` defines a keyword that takes on one or more enumerated values with minimum-match string expansion.
- `option()` defines one or more options.
- `custom()` defines a keyword that is handled in a custom way by the caller.

The object returned by `process()` has an attribute for each keyword defined using the above functions.

- For keywords defined with `keyword()`, the attribute is equal to the user's provided value or the default.

- For keywords defined with `mkeyword()`, the attribute is a list of the values provided by the user, with the list being empty if the user provided none.
- For keywords defined with `keymatch()`, the attribute is a list of the values provided by the user expanded out to their full values if the user abbreviated any. As above, the list may be empty.
- For options defined with `option()`, the attribute is either `True` or `False`.
- For keywords defined with `custom()`, the attribute is whatever value was returned by a user-specified routine.

If the user-specified values do not match the expectations defined by the specification (e.g., a keyword that should be integer-typed is passed the value "abcd") then a `MiriadError` is raised in `process()`.

Keyword Types

Keyword types in MIRIAD and miriad-python are identified by single letters. The following types are available:

Character	Description
i	(The character is the lower-case letter i.) An integer value.
d	A floating-point ("double") value.
a	A character string value.
f	A filename value. These are essentially treated like character strings, but there are special hooks in the MIRIAD processing code to expand out shell glob patterns into multiple values.
t	A time or angle value. These are parsed according to one of several formats, described below. The output is always a floating-point number but its meaning depends on the parse format.

Keyword Formats

Keywords describing a time or angle can be parsed according to one of several formats. You must specify one of these formats when declaring the keyword.

Name	Description
dms	The argument is an angle measured in degrees, written as "dd:mm:ss.s" or "dd.ddd". The output is the angle in radians.
hms	The argument is an angle/time measured in hours, written as "hh:mm:ss.s" or "hh.hhh". The output is the angle/time in radians.
dtime	The argument is a day fraction, i.e. the portion of a day that has elapsed at that local time. The user can provide it in the format "hh:mm:ss.s" or "hh.hhhh". The output is the day fraction, a number in the range [0, 1].
atime	The argument is an absolute time, specified as either "yyMMMdd.ddd" or "yyMMMdd:hh:mm:ss.s", or as an epoch, "bYYYY" or "jYYYY". The output is in Julian days.
time	Either an absolute time or a day fraction. The output is either a Julian day value or a day fraction.

Integration with the UVDAT Subsystem

The keyword subsystem can integrate with MIRIAD's UVDAT subsystem. This integration is necessary because the UVDAT subsystem uses the MIRIAD keyword-handling routines to obtain settings for UV data selection, whether calibration should be applied, and so on.

If your task does not use the UVDAT subsystem, you need take no action.

If your task does use the UVDAT subsystem, you must call `KeySpec.uvdat()` while defining your keywords. When doing so, you specify processing options that will be passed to the UVDAT subsystem. See the documentation of `uvdat()` for more information.

`mirtask.keys` API Reference

This section presents a detailed API reference for the *`mirtask.keys`* module.

1.4.2 The UVDAT Subsystem for Reading UV Data: `mirtask.uvdat`

Foo.

`mirtask.uvdat` API Reference

This section presents a detailed API reference for the *`mirtask.uvdat`* module.

1.4.3 Utilities for Command-Line Programs: `mirtask.cliutil`

CHAPTER 2

Indices and Tables

- genindex
- modindex
- search

CHAPTER 3

Colophon

This documentation describes *miriad-python* version 1.2.4, implementing version 1.2 of the API. It was generated on June 26, 2023. This documentation is created using Sphinx.

Python Module Index

m

Index