

# sidereal.py: A Python package for astronomical calculations



John W. Shipman

2010-01-07 15:07



## Table of Contents

1. Introduction .....	2
2. Downloadable files .....	3
3. Coordinate systems used .....	3
3.1. Terrestrial coordinates .....	3
3.2. Altazimuth coordinates .....	3
3.3. Equatorial coordinates .....	4
4. Standalone scripts .....	5
4.1. Specifying dates and times on command lines .....	5
4.2. Specifying angles on command lines .....	6
4.3. Specifying latitude and longitude on command lines .....	6
4.4. Specifying hours on command lines .....	6
4.5. jd: Convert to Julian date .....	7
4.6. conjd: Convert from Julian date .....	7
4.7. rdaa: Equatorial to horizon coordinates .....	7
4.8. aard: Horizon to equatorial coordinates .....	8
5. Constants .....	8
6. Functions .....	9
6.1. hoursToRadians(): Convert hours to radians .....	9
6.2. radiansToHours(): Convert radians to hours .....	9
6.3. hourAngleToRA(): Convert an hour angle to right ascension .....	9
6.4. raToHourAngle(): Convert a right ascension to an hour angle .....	9
6.5. dayNo(): Date to day number .....	10
6.6. parseDatetime(): Parse external date/time .....	10
6.7. parseDate(): Parse external date .....	10
6.8. parseTime(): Parse external time .....	10
6.9. parseAngle(): Parse external angle .....	10
6.10. parseLat(): Parse latitude .....	11
6.11. parseLon(): Parse longitude .....	11
6.12. parseHours(): Parse a quantity in hours .....	11
7. class MixedUnits: Handling mixed unit systems .....	11
7.1. MixedUnits.__init__(): Constructor .....	12
7.2. MixedUnits.mixToSingle(): Convert mixed units to a single value .....	12
7.3. MixedUnits.singleToMix(): Convert a single value to mixed units .....	12
7.4. MixedUnits.format(): Format with truncation .....	13
8. dmsUnits: Converter to and from degrees, minutes, and seconds .....	13
9. class LatLon: Terrestrial position .....	13

9.1. LatLon.__init__(): Constructor .....	13
9.2. LatLon.__str__(): Convert to a string .....	14
10. class JulianDate: Julian date and time .....	14
10.1. JulianDate.__init__(): Constructor .....	14
10.2. JulianDate.__float__(): Convert to float .....	15
10.3. JulianDate.datetime(): Convert to a datetime instance .....	15
10.4. JulianDate.offset(): Move a time by some number of days .....	15
10.5. JulianDate.__sub__(): Find the difference between two times .....	15
10.6. JulianDate.__cmp__(): Comparison .....	15
10.7. JulianDate.fromDatetime(): Convert a datetime .....	15
11. class SiderealTime .....	15
11.1. SiderealTime.__init__(): Constructor .....	16
11.2. SiderealTime.__str__(): Convert to a string .....	16
11.3. SiderealTime.utc(): Find Universal Time .....	16
11.4. SiderealTime.gst(): Local Sidereal Time to Greenwich Sidereal Time .....	16
11.5. SiderealTime.lst(): Greenwich Sidereal Time to Local Sidereal Time .....	16
11.6. SiderealTime.fromDatetime(): Convert UTC to GST .....	16
12. class AltAz: Horizon coordinates .....	17
12.1. AltAz.__init__(): Constructor .....	17
12.2. AltAz.raDec(): Convert horizon to equatorial coordinates .....	17
12.3. AltAz.__str__(): Convert to a string .....	17
13. class RADEC: Equatorial coordinates .....	17
13.1. RADEC.__init__(): Constructor .....	18
13.2. RADEC.hourAngle(): Find the hour angle of an equatorial coordinate .....	18
13.3. RADEC.altAz(): Convert to horizon coordinates .....	18
13.4. RADEC.__str__(): Convert to a string .....	18

This document describes `sidereal.py`, a Python module to perform spherical geometry calculations for astronomical applications.

This publication is available in Web form<sup>1</sup> and also as a PDF document<sup>2</sup>. Please forward any comments to **tcc-doc@nmt.edu**.

## 1. Introduction

---

The `sidereal.py` package is designed to do a few of the simpler astronomical calculations. The mathematical basis of these calculations comes from spherical trigonometry.

Here is the source for formulae used in this package:

Duffett-Smith, Peter. Practical astronomy with your calculator. Second edition. Cambridge, 1981, ISBN 0-521-28411-2.

The reader is assumed to know the Python programming language.

### Caution

All astronomical computations are approximations. Refer to Duffett-Smith's book for a discussion of the limitations of these formulae. In particular, they assume that the Earth is a sphere, which it isn't quite.

<sup>1</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/sidereal/>

<sup>2</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/sidereal/sidereal.pdf>

The actual implementation of this module is shown in the companion document, *sidereal.py: Internal maintenance specification*<sup>3</sup>, in lightweight literate programming form<sup>4</sup>.

## 2. Downloadable files

---

- `sidereal.py`<sup>5</sup>: The Python module
- `jd`<sup>6</sup>: Civil date to Julian date
- `conjd`<sup>7</sup>: Julian date to civil date
- `rdaa`<sup>8</sup>: Equatorial to horizon coordinates
- `aard`<sup>9</sup>: Horizon to equatorial coordinates

## 3. Coordinate systems used

---

We must first define how we point at locations on the earth's surface and in the sky.

- Section 3.1, "Terrestrial coordinates" (p. 3).
- Section 3.2, "Altazimuth coordinates" (p. 3).
- Section 3.3, "Equatorial coordinates" (p. 4).

### 3.1. Terrestrial coordinates

Earth is assumed to be a sphere, and any location can be specified using two numbers.

- The *longitude* is the angle relative to the Greenwich Meridian, a line drawn from the South Pole to the North Pole and passing through a certain point in London. For convenience in astrometrical calculations, we normalize them to the range  $[0, 2\pi)$ , with values increasing to the east. This is contrary to the usual geographic convention of placing them in the range  $[-\pi, \pi]$ .
- The *latitude* is the angle relative to the equator, which is in the plane of the earth's rotation. North latitudes are positive, southern latitudes negative. In radians, the values lie in the interval  $[-\pi/2, \pi/2]$ .

### 3.2. Altazimuth coordinates

One way to describe the location of an object in the sky is by using *altazimuth coordinates*, also known as *horizon coordinates*. These coordinates are always relative to a specific observer's location.

It takes two numbers to describe a location in horizon coordinates.

- The *altitude* or *elevation* is the height of the object above the idealized horizon (or where the horizon would be if the local terrain were flat). An elevation of  $\pi/2$  radians is straight up, also known as the *zenith*.

In practice, some objects at negative altitudes are visible: from South Baldy at 10,000' in the Magdalenas, it is possible to see a bit below the horizon in some directions.

---

<sup>3</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/sidereal/ims/>

<sup>4</sup> <http://www.nmt.edu/~shipman/soft/litprog/>

<sup>5</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/sidereal/ims/sidereal.py>

<sup>6</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/sidereal/ims/jd>

<sup>7</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/sidereal/ims/conjd>

<sup>8</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/sidereal/ims/rdaa>

<sup>9</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/sidereal/ims/aard>

- The *azimuth* is the equivalent of the compass direction. An object due north of the observer is considered azimuth zero, and values increase clockwise in the order N→E→S→W→N. In radians, the values are in the range  $[0, 2\pi)$ .

### 3.3. Equatorial coordinates

The vast majority of objects in the sky, such as distant galaxies, are fixed relative to each other, for all practical purposes. We call this reference frame the *sidereal* reference frame.

By convention, coordinates in this reference frame are specified relative to the earth's plane of rotation. The *celestial equator* is the line around the celestial sphere that is directly overhead if you are standing on the earth's equator. So sidereal coordinates are also called *equatorial coordinates*. It takes two numbers to specify a location in the sidereal frame.

- The *declination* is the angle relative to the celestial equator. Positive angles are north of the equator.
- The *right ascension* is the angle relative to a reference point called the *first point of Aries*. The right ascension increases toward the east.

Right ascension is in units of *hours*, where an hour is defined as 15 degrees. Therefore, the right ascension of an object in celestial coordinates is in the half-open interval  $[0, 24)$ .

Below are some other important terms defined relative to sidereal positions.

#### 3.3.1. First point of Aries

This point is the origin of the equatorial coordinate system. It corresponds to right ascension zero, declination zero. It lies at the intersection of the plane of the earth's rotation with the plane of the earth's orbit around the Sun.

#### 3.3.2. Vernal equinox

Another name for the first point of Aries.

#### 3.3.3. The celestial poles

If we project the axis of the earth's rotation into the sky, the north and south celestial poles are the points where that axis intersects the sky.

Celestial objects appear to move in circles centered on these points. The north celestial pole has a declination of  $+\pi/2$ , and the south celestial pole is at declination  $-\pi/2$ .

#### 3.3.4. The meridian

From a given observer's position, the meridian is the line in the sky that passes through azimuth zero (north), the zenith (altitude  $90^\circ$ ), and azimuth  $180^\circ$  (south).

#### 3.3.5. Transit

*Transit* is defined as the time at which an object at a given right ascension crosses the meridian.

### 3.3.6. Hour angle

For a given celestial position, its hour angle is zero when that position transits the meridian. As that position moves from east to west in the sky, the hour angle increases at a rate of about  $15^\circ$  per hour until it wraps around from  $2\pi$  to zero at its next transit.

### 3.3.7. Sidereal time

Sidereal time is defined as the hour angle of the first point of Aries. Essentially, it describes a specific position of the sky relative to the earth.

*Local sidereal time* (LST) is the sidereal time for a given observer's position on the earth. Whenever the sidereal time is the same, the sky will look the same from that position.

*Greenwich sidereal time* (GST) is the sidereal time at longitude  $0^\circ$ .

Because the year is not an exact multiple of the earth's rotational period, the sidereal day is about four minutes shorter than 24 hours. Therefore, the sidereal times during the last four minutes before midnight duplicate that same range of sidereal times at the beginning of the day.

## 4. Standalone scripts

---

These scripts are used to perform some common operations from the Unix command line:

- Section 4.5, “`jd`: Convert to Julian date” (p. 7).
- Section 4.6, “`conj d`: Convert from Julian date” (p. 7).
- Section 4.7, “`rdaa`: Equatorial to horizon coordinates” (p. 7).
- Section 4.8, “`aard`: Horizon to equatorial coordinates” (p. 8).

These scripts all require you to express times, angles, and similar units on the command line. Please refer to these sections for these common notational conventions:

- Section 4.1, “Specifying dates and times on command lines” (p. 5).
- Section 4.2, “Specifying angles on command lines” (p. 6).
- Section 4.3, “Specifying latitude and longitude on command lines” (p. 6).
- Section 4.4, “Specifying hours on command lines” (p. 6).

### 4.1. Specifying dates and times on command lines

The scripts in this package require dates, with or without times, in this format, which is taken from the ISO standard date/time format:

```
YYYY-MM-DD[Thh[:mm[:ss.sss...]]]zone
```

The “T” separating the date and time is not case-sensitive.

Without the *zone* specifier, the time is assumed to be UTC (Greenwich time). You may also specify these zones (not case-sensitive):

$\pm hhmm$	Hours and minutes east (+) or west (-) of UTC. Example: -0700 for Mountain Standard Time.
UTC	Universal Coordinated Time, roughly Greenwich Mean.
EST	Eastern Standard Time, -0500.
EDT	Eastern Daylight Time, -0400.

ET	Eastern Time: either EST or EDT depending on the date. At this time, the package supports the U.S. rules for daylight time, assuming that dates before 2007 change on the first Sunday in April and the last Sunday of October, and that 2007 and later change on the second Sunday in March and the first Sunday in November.
CST, CDT, CT	Central Standard (-0600), Central Daylight (-0500), and Central Time, with the same rules for daylight time as ET.
MST, MDT, MT	Mountain Standard (-0700), Mountain Daylight (-0600), and Mountain Time.
PST, PDT, PT	Pacific Standard (-0800), Pacific Daylight (-0700), and Pacific Time.

Here are some examples:

```
1918-05-07
2004-02-29T06utc
2004-02-29t06:18:32.92287mt
2004-02-29T06:18+0300
```

## 4.2. Specifying angles on command lines

In the human world of the command line, angles will always be represented in degrees, optionally with minutes and seconds, in one of these forms:

```
NN.NNNd
NNdNN.NNNm
NNdNNmNN.NNNs
```

Examples:

```
0.0036782d
14d08m
14d08.37m
131d50m03.0017s
```

## 4.3. Specifying latitude and longitude on command lines

To specify an observer's location, humans will want to work in latitude and longitude in degrees, optionally with minutes and seconds. Latitudes are expressed as an angle followed by letter **N** or **S** (case-insensitive); longitudes are expressed as an angle followed by letter **E** or **W**. For the syntax of angles, see Section 4.2, "Specifying angles on command lines" (p. 6).

Examples:

```
34d08m42sn 107d50m37sw
41.86333N 78.961389W
```

## 4.4. Specifying hours on command lines

Right ascension, hour angle, and sidereal time are all expressed in hours. The syntax for such quantities is pretty similar to the syntax for angles, except that the first unit has an "h" suffix for hours.

```
NN.NNNh
NNhNN.NNNm
NNhNNmNN.NNNs
```

Examples:

```
05h51m44s
05H51M44S
05.86222h
```

## 4.5. `jd`: Convert to Julian date

To find the Julian date number for a given date, with or without a time of day, use a command of this form:

```
jd datetime
```

For the syntax of dates and times, see Section 4.1, “Specifying dates and times on command lines” (p. 5). As a convenience, you can supply the date and time as two separate arguments without the “T” separator between them.

Examples:

```
$ jd 1985-02-17t06:00
2446113.75
$ jd 1985-02-17 6:00
2446113.75
$
```

## 4.6. `conjd`: Convert from Julian date

To express a Julian date in conventional units, use this script:

```
conjd JD
```

The *JD* argument is the Julian date. Example:

```
$ conjd 2446113.75
1985-02-17 06:00:00
```

## 4.7. `rdaa`: Equatorial to horizon coordinates

To find the position in some observer's sky of a celestial object with right ascension *RA*, and declination *DEC*, use this command:

```
rdaa RA±DEC lat lon dt
```

### ***RA±DEC***

Right ascension in hours, followed by the signed declination; either “+” or “-” is required as a separator. For the format of hour quantities, see Section 4.4, “Specifying hours on command lines” (p. 6); for the format of the declination, see Section 4.2, “Specifying angles on command lines” (p. 6).

**lat lon**

The next two arguments are the observer's latitude and longitude, respectively; see Section 4.3, "Specifying latitude and longitude on command lines" (p. 6) for the syntax of these arguments.

**dt**

The date and time of the observation, using the syntax described in Section 4.1, "Specifying dates and times on command lines" (p. 5).

## 4.8. aard: Horizon to equatorial coordinates

To convert a celestial position in horizon coordinates to the equivalent equatorial coordinates for a given observer's precision, use a command of this form:

```
aard az±alt lat lon dt
```

**az±alt**

Azimuth of the object, followed by either "+" or "-" sign and altitude. For the syntax of these quantities, see Section 4.2, "Specifying angles on command lines" (p. 6).

**lat lon**

Latitude and longitude of the observer; see Section 4.3, "Specifying latitude and longitude on command lines" (p. 6) for their formats.

**dt**

Date and time of the observation; see Section 4.1, "Specifying dates and times on command lines" (p. 5).

This example would find the right ascension and declination of a celestial object viewed at an azimuth of 283°16'18" and an altitude of 19°20'2", with the observer located at 34°8'42" N, 107°50'37" W, on Feb. 29, 2008, at 18:46 Mountain Time.

```
aard 283d16m18s+19d20m2s 34d08m42sn 107d50m37sw 2008-02-29t18:46mt
```

## 5. Constants

---

These constants are exported by the `sidereal.py` module.

**FIRST\_GREGORIAN\_YEAR**

The year when the Gregorian calendar took over from the flawed Julian calendar: 1583.

**PI\_OVER\_180**

The value of  $(\pi/180)$ . Multiply degrees by this value to get radians.

**TWO\_PI**

The value of  $(2\pi)$ . Used for normalizing angles.

**PI\_OVER\_12**

The value of  $(\pi/12)$ . Multiply hours by this value to get radians.

**JULIAN\_BIAS**

A constant amount removed from Julian dates for internal storage, to allow for more precision in times of day. For a discussion, see Section 10, "class JulianDate: Julian date and time" (p. 14).

## 6. Functions

---

These simple functions are exported by the `sidereal.py` module.

### 6.1. `hoursToRadians()`: Convert hours to radians

```
hoursToRadians ( hours )
```

The argument is in hours, that is,  $15^\circ$ ; units. The result is the same angle expressed in radians.

### 6.2. `radiansToHours()`: Convert radians to hours

The inverse function of Section 6.1, “`hoursToRadians()`: Convert hours to radians” (p. 9), it returns the equivalent number of  $15^\circ$  hours.

```
radiansToHours ( radians )
```

### 6.3. `hourAngleToRA()`: Convert an hour angle to right ascension

This function converts an hour angle to the equivalent right ascension (in radians) for a specific time and location. For definitions, see Section 3.3.6, “Hour angle” (p. 5).

```
hourAngleToRA(h, ut, eLong)
```

**h**

The hour angle in radians.

**ut**

The Universal Time as a `datetime` instance.

**eLong**

The east longitude in radians.

### 6.4. `raToHourAngle()`: Convert a right ascension to an hour angle

This is the inverse of Section 6.3, “`hourAngleToRA()`: Convert an hour angle to right ascension” (p. 9): given a right ascension `ra` and the observer's time and location, it returns the hour angle in radians.

```
raToHourAngle(ra, ut, eLong)
```

**ra**

The right ascension in radians.

**ut**

The Universal Time as a `datetime` instance.

**eLong**

The east longitude in radians.

## 6.5. `dayNo ( )`: Date to day number

This function takes a date `dt`, as either a `datetime.date` instance or a `datetime.datetime` instance, and returns the day number relative to January 0 of that year. For example, the day number of January 1 is 1, and the day number of December 31 is either 365 or 366.

```
dayNo ( dt )
```

## 6.6. `parseDatetime ( )`: Parse external date/time

This function converts an external timestamp in the form described in Section 4.1, “Specifying dates and times on command lines” (p. 5).

```
parseDatetime ( s )
```

The argument `s` is a string containing the timestamp string. If it is valid, the function returns that timestamp as a `datetime.datetime` instance. If invalid, it will raise a `SyntaxError` exception.

## 6.7. `parseDate ( )`: Parse external date

This function converts an external date in the form described in Section 4.1, “Specifying dates and times on command lines” (p. 5). The date must not be followed by “T” and a time.

```
parseDate ( s )
```

Argument `s` is the string to be converted. If successful, the function returns the date as a `datetime.date` instance. If the string is not a valid date, the function raises a `SyntaxError` exception.

## 6.8. `parseTime ( )`: Parse external time

This function converts a time of day, with optional time zone suffix. For the syntax, see Section 4.1, “Specifying dates and times on command lines” (p. 5); the time is everything after the “T” separator.

```
parseTime ( s )
```

The argument `s` is a string whose format must match the specified format. If successful, the function returns the time as a `datetime.time` instance. If the format is incorrect, the function raises a `SyntaxError` exception.

## 6.9. `parseAngle ( )`: Parse external angle

This function parses an angle in the form described in Section 4.2, “Specifying angles on command lines” (p. 6).

```
parseAngle ( s )
```

Argument `s` is the string to be parsed. If successful, the function returns the angle in *radians*. It raises a `SyntaxError` exception if the format is incorrect.

## 6.10. parseLat ( ): Parse latitude

This function parses a latitude string in the form described in Section 4.3, “Specifying latitude and longitude on command lines” (p. 6).

```
parseLat ( s )
```

If the argument `s` is a valid latitude string in degrees, it is returned as a signed value in radians in the interval  $[-\pi, \pi]$ . It will raise a `SyntaxError` if `s` does not have the correct format.

## 6.11. parseLon ( ): Parse longitude

This function parses a longitude string in the form described in Section 4.3, “Specifying latitude and longitude on command lines” (p. 6).

```
parseLon ( s )
```

If the argument `s` is a valid longitude string in degrees, it is returned as a value in radians in the interval  $[0, 2\pi)$ . It will raise a `SyntaxError` if `s` does not have the correct format.

## 6.12. parseHours ( ): Parse a quantity in hours

This function parses a string containing a value in hours with optional minutes and seconds, as described in Section 4.4, “Specifying hours on command lines” (p. 6).

```
parseHours ( s )
```

If string `s` is valid, the function returns the angle as a float in interval  $[0,24)$ . It will raise a `SyntaxError` if `s` is not valid.

# 7. class MixedUnits: Handling mixed unit systems

---

The `sidereal.py` package uses a number of systems of mixed units:

- Degrees, minutes, and seconds for latitudes and longitudes.
- Hours, minutes, and seconds, for right ascensions and hour angles.
- Hours, minutes, and seconds for times of day and sidereal times.

Operations on mixed-unit quantities include:

- Converting from mixed units to a single unit. For example, converting an angle like  $38^\circ 52' 30.7''$  to decimal degrees.
- Converting a single quantity to mixed units. For example, convert 14.876 hours to hours, minutes, and seconds.
- Formatting a single quantity as mixed units. This is not as straightforward as it seems! If you just convert to mixed units and format each number, there is an ugly pathology for certain values. For example, suppose you have a tuple representing the angle  $12^\circ 13' 59.999''$ , like this:

```
>>> angle = (12, 13, 59.999)
```

And then you format these quantities, using only one digit after the decimal in the seconds term, watch what happens:

```
>>> print '%dd %d\' %.1f"' % angle
12d 13' 60.0"
```

That is not user-friendly. We should display it either as “12d 14' 0.0” or “12d 13' 59.9”.

The class interface shown below has a special method, `.format()`, that prevents this problem.

In order to make calculations on mixed units, it is necessary to define the relative size of the units in the system. For example, in the days-hours-minutes-seconds system, there are 24 hours in a day, 60 minutes in an hour, and 60 hours in a second.

So we'll define a sequence called the *factor list* as a Python sequence containing these factors. For example, the factor list for the days-hours-minutes-seconds system is (24, 60, 60). The factor list for a mixed unit system with  $N$  units always has length  $(N-1)$ .

## 7.1. `MixedUnits.__init__()`: Constructor

To create a `MixedUnits` instance, pass the factor list to its constructor like this:

```
MixedUnits ( factorList )
```

For example, here is how you would create an instance to represent the days-hours-minutes-seconds mixed units system.

```
dhms = MixedUnits ( (24, 60, 60) )
```

Each instance has an attribute `.factors` containing the factor list passed to the constructor.

## 7.2. `MixedUnits.mixToSingle()`: Convert mixed units to a single value

For a `MixedUnits` instance  $M$ , this method takes a sequence of values representing mixed units, and converts them to a single value in the largest unit.

```
M.mixToSingle ( valueList )
```

Here's an example. In the degrees-minutes-seconds system, these statements will convert 38° 52' 30.7" to decimal degrees.

```
>>> dmsSystem = MixedUnits ( (60, 60) )
>>> degrees = dmsSystem.mixToSingle ( (38, 52, 30.7) )
>>> print degrees
38.875194
```

If the `valueList` is shorter than the maximum for the system, units are assumed to be from the large end of the system. For example, in the degrees-minutes-seconds system, the value (19, 17.3) is assumed to be 19° 17.3'.

## 7.3. `MixedUnits.singleToMix()`: Convert a single value to mixed units

This is the inverse of the `.mixToSingle()` method. For a `MixedUnits` instance  $M$ , it takes a single value, in the largest unit in the system, and converts it to a sequence of units in mixed form.

```
M.mixToSingle ( value )
```

For example, this code would convert 14.876 hours to hours, minutes and seconds, setting `hms` to a 3-element tuple.

```
hmsSystem = MixedUnits ( (60, 60) )
hms = hmsSystem.singleToMix ( 14.876 )
```

## 7.4. `MixedUnits.format()`: Format with truncation

For a `MixedUnits` instance  $M$ , and a sequence `mix` like the one returned by the `.singleToMix()` method, this method returns formatted values as a list of strings. All but the last value will be displayed as integers. You can specify how many digits of precision will be displayed after the decimal point in the last value as argument `decimals`, which defaults to zero.

```
M.format ( mix, decimals=0, lz=False )
```

By default, values take only as many digits as they need. If you would like all the values left-zero-padded to have at least two digits before the decimal, pass `lz=True`.

For example, this would format the mixed-unit angle  $12^\circ 13' 59.999''$  with one digit after the decimal point of the seconds string:

```
>>> dmsSystem = MixedUnits ( (60, 60) )
>>> print dmsSystem.format ( (12, 13, 59.999), 1 )
['12', '13', '59.9']
>>> print dmsSystem.format ( (1,2,3), 3, lz=True )
['01', '02', '03.000']
```

It could be argued that the result in this case should be `['12', '14', '0.0']`. The package will opt for truncation instead of rounding. If this is a problem, consider using more precision.

## 8. `dmsUnits`: Converter to and from degrees, minutes, and seconds

---

Global to this module is an instance of `MixedUnits` named `dmsUnits`, which is a facility to allow conversion to and from two mixed-unit systems used throughout: hours/minutes/seconds and degrees/minutes/seconds, which happen to have the same factor list, namely, `(60, 60)`.

For example, to convert 14.876 hours to mixed units and back:

```
>>> dmsUnits.singleToMix(14.876)
(14, 52, 33.59999999999998005)
>>> dmsUnits.mixToSingle(_)
14.876
```

## 9. class `LatLon`: Terrestrial position

---

An instance of this class represents some point on the earth's surface.

### 9.1. `LatLon.__init__()`: Constructor

To create a `LatLon` instance:

```
LatLon ( lat, lon )
```

**lat**

The latitude in radians, in the range  $[-\pi, \pi]$ .

**lon**

The longitude in radians. Values will be normalized to the range  $[0, 2\pi)$ .

Instance attributes `.lat` and `.lon` will contain the arguments passed to the constructor.

## 9.2. LatLon.\_\_str\_\_(): Convert to a string

Applying the standard Python `str()` function to a `LatLon` instance will return a string displaying the latitude and longitude in the usual human units, e.g., "03d 14m 32s N Lat 118d 32m 04s W Lon".

## 10. class JulianDate: Julian date and time

---

The Julian date is the most common way of expressing the time at which an astronomical event occurs. It is independent of the time zone. It is defined as the number of days since noon on January 1, 4713 BC at the Greenwich meridian. Current Julian dates have seven digits to the left of the decimal point: for example, noon on January 1, 2000 is Julian date 2,451,545.0.

Your Python program can handle a `JulianDate` instance as if it were an ordinary floating point number. However, since IEEE float values are good only to about 15 digits of precision, this means that the precision of times of day is on the order of  $1 \times 10^{-8}$  days, or in the neighborhood of a millisecond.

Assuming that modern astronomy is more important than historical research, we can buy a bit more precision by storing internally the Julian date minus 2,200,000, which gives us at least 9 or 10 digits after the decimal, or a precision of roughly 10 microseconds.

Through the magic of Python classes, though, converting a `JulianDate` instance into a Python `float` will restore the bias, giving current exact values. However, subtraction of two `JulianDate` values will cause less of a problem with loss of significance than subtracting two `float` values converted from `JulianDate` instances.

Here is the class interface.

### 10.1. JulianDate.\_\_init\_\_(): Constructor

```
JulianDate(j, f=0.0)
```

**j**

The Julian date as a `float` or `int`.

**f**

If you would like to work with greater precision, pass the integral part of the Julian date in as `j` and the fractional part as `f`. The bias (`JULIAN_BIAS`) will be removed from `j` before `f` is added, giving you extended precision.

Instance attribute `.j` contains the biased value, that is, `j+f - JULIAN_BIAS`.

## 10.2. `JulianDate.__float__()`: Convert to float

To convert a `JulianDate` instance back to a `float`, use the regular Python `float()` function.

## 10.3. `JulianDate.datetime()`: Convert to a `datetime` instance

This method converts an instance *J* of class `JulianDate` into an instance of the `datetime` class from the standard Python library (version 2.3 or greater).

```
J.datetime()
```

You will get back a “naive `datetime` instance” tagged with no time zone information, but it will represent UTC.

## 10.4. `JulianDate.offset()`: Move a time by some number of days

To obtain a new `JulianDate` instance that differs from some instance *J* by some number of days *D*, use this method:

```
J.offset(D)
```

For example, to get a date three days in the past, use a *D* value of -3.

## 10.5. `JulianDate.__sub__()`: Find the difference between two times

The ordinary Python subtract operator works on `JulianDate` instances. It returns the difference in days between the two values as a `float`.

## 10.6. `JulianDate.__cmp__()`: Comparison

The ordinary Python comparison operators are implemented for `JulianDate` instances.

## 10.7. `JulianDate.fromDatetime()`: Convert a `datetime`

Use this static method to convert a standard Python `datetime` instance into a `JulianDate` instance.

```
JulianDate.fromDatetime(dt)
```

The *dt* argument is a `datetime` instance. If *dt* contains time zone information, the returned `JulianDate` instance will represent the corresponding UTC. If *dt* is “naive”, containing no time zone information, *it is assumed to represent UTC*.

# 11. class `SiderealTime`

An instance of this class represents a sidereal time in hours. For definitions, see Section 3.3.7, “Sidereal time” (p. 5).

These instance attributes are exported:

### **.hours**

The sidereal time as decimal hours in the range [0,24).

## **.radians**

The sidereal time in radians, in the range  $[0, 2\pi)$ .

### **11.1. SiderealTime.\_\_init\_\_(): Constructor**

The constructor takes a value  $S$  in the interval  $[0, 24)$ .

```
SiderealTime(S)
```

### **11.2. SiderealTime.\_\_str\_\_(): Convert to a string**

When converted to strings, sidereal times will be displayed in this general form:

```
[HHh MMm SS.SSSs]
```

### **11.3. SiderealTime.utc(): Find Universal Time**

For a `SiderealTime` instance  $S$ , and a `date` instance  $D$ , this method returns a `datetime` instance representing the first or only time on that date when the sidereal time is  $S$ .

```
S.utc(D)
```

### **11.4. SiderealTime.gst(): Local Sidereal Time to Greenwich Sidereal Time**

For a `SiderealTime` instance  $LST$  representing local sidereal time at some longitude  $eLong$  (in radians), this function returns a `SiderealTime` instance representing Greenwich Sidereal Time (GST).

```
LST.gst(eLong)
```

### **11.5. SiderealTime.lst(): Greenwich Sidereal Time to Local Sidereal Time**

For a `SiderealTime` instance  $GST$  representing Greenwich Sidereal Time, this function returns a `SiderealTime` instance representing local sidereal time at some longitude  $eLong$  (in radians).

```
GST.lst(eLong)
```

### **11.6. SiderealTime.fromDatetime(): Convert UTC to GST**

This static method takes a standard Python `datetime` instance and returns a new `SiderealTime` instance representing the corresponding Greenwich Sidereal Time.

```
SiderealTime.fromDatetime(dt)
```

The argument is a `datetime` instance. If  $dt$  is an “aware” timestamp instance, with time zone information, the result represents the sidereal time at the equivalent UTC corresponding. If  $dt$  is “naive,” with no time zone information, it is assumed to represent UTC.

## 12. class AltAz: Horizon coordinates

---

An instance of class `AltAz` represents a location in the sky relative to the observer's horizon. See Section 3.2, “Altazimuth coordinates” (p. 3).

### 12.1. AltAz.\_\_init\_\_(): Constructor

The constructor call has this form:

```
AltAz ( alt, az )
```

#### **alt**

The altitude above the horizon in radians.

#### **az**

The azimuth angle in radians, where 0.0 is north,  $\pi/2$  is east, and so on.

Instances have these attributes:

#### **.alt**

The altitude, normalized to  $[-\pi, \pi]$ .

#### **.az**

The azimuth, normalized to  $[0, 2\pi)$ .

### 12.2. AltAz.raDec(): Convert horizon to equatorial coordinates

This method performs one of the fundamental astronomical calculations: converting a position in the sky expressed in horizon coordinates to the corresponding position in equatorial coordinates.

To do this conversion, we must know the time and place. For an `AltAz` instance `AA`, this method returns a position in equatorial coordinates as a `RADec` instance.

```
AA.raDec(lst, latLon)
```

#### **lst**

The local sidereal time at the observer's location, as a `SiderealTime` instance.

#### **latLon**

The observer's terrestrial position, as a `LatLon` instance.

### 12.3. AltAz.\_\_str\_\_(): Convert to a string

Applying the Python `str()` function to an `AltAz` instance returns a string of this form:

```
[az NNnd NN' NN.NNN" alt NNnd NN' NN.NNN"]
```

where  $X$  is the altitude and  $Y$  is the azimuth, both shown in degrees.

## 13. class RADec: Equatorial coordinates

---

An instance of this class represents a celestial position in equatorial coordinates; see Section 3.3, “Equatorial coordinates” (p. 4).

### 13.1. `RADec.__init__()`: Constructor

The class constructor has this form:

```
RADec ( ra, dec )
```

**ra**

The right ascension coordinate in radians.

**dec**

The declination coordinate in radians.

### 13.2. `RADec.hourAngle()`: Find the hour angle of an equatorial coordinate

For an `RADec` instance *RD*, this method returns that position's hour angle in radians for a given observer's position and time.

```
RD.hourAngle(ut, eLong)
```

**ut**

The Universal Time at the observer's position as a `datetime` instance.

**eLong**

The longitude of the observer's position in radians.

### 13.3. `RADec.altAz()`: Convert to horizon coordinates

For an `RADec` instance *RD*, this method returns the equivalent horizon coordinates for a given observer's position and time, as an `AltAz` instance.

```
RD.altAz(h, lat)
```

**h**

The hour angle of the celestial position.

**lat**

The observer's latitude in radians.

### 13.4. `RADec.__str__()`: Convert to a string

Applying the Python `str()` function to an `RADec` instance returns a string showing the right ascension in hours and the declination in degrees, in this form:

```
[ddh ddm dd.ddds, ddd dd' dd.ddd"]
```