Astroinformatics I (Semester 1 2024)

Python Astronomical Packages

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

June 24, 2024

Motivation

We have seen so far that Python provides a huge variety of scientific packages and libraries, e.g. numpy, SciPy and matplotlib.

In some cases, however, we need functionality that is specifically related to astronomy and astrophysics.

We look here into **astronomical packages**, especially `astropy`. We have already seen `astropy` in how to retrieve astronomical data. The package, however, also provides plenty other relevant functionality, such as constants, units, time conversion and much more.

Units and Constants

Despite calculations are often done without keeping the units, it makes sense to keep track of the units:

Was the wavelength given in Angstrom or in nm? In X-ray observations, a common unit of wavelength is keV. How many nm is 0.65 keV?
Are data from multiple surveys incorporated?

Units and Constants

Despite calculations are often done without keeping the units, it makes sense to keep track of the units:

Was the wavelength given in Angstrom or in nm? In X-ray observations, a common unit of wavelength is keV. How many nm is 0.65 keV?
Are data from multiple surveys incorporated?

`astropy.units` offers a framework that can take care of this and propagates the units through many (but not all) mathematical operations (e.g. addition, division, multiplication).

Furthermore, `astropy.constants` supplies the values of many physical and astronomical constants.

Units and Constants

The easiest way to attach a **unit** to a number is by multiplication:

Most users of the astropy.units package will work with Quantity objects: the combination of a value and a unit. The most convenient way to create a Quantity is to multiply or divide a value by one of the built-in units. It works with scalars, sequences, and numpy arrays.

```
>>> from astropy import units as u

>>> 42.0 * u.meter
<Quantity  42. m>


>>> [1., 2., 3.] * u.m
<Quantity [1., 2., 3.] m>
```

Units and Constants

One can get the unit and value from a `Quantity` using the unit and value members:

```
>>> q = 42.0 * u.meter

>>> q.value
42.0

>>> q.unit
Unit("m")
```

Unit conversion is done using the `to()` method, which returns a new `Quantity` in the given unit:

```
x = 1.0 * u.parsec

x.to(u.km)
<Quantity 30856775814671.914 km>
```

Units and Constants

Using this basic building block, it is possible to combine quantities with different units. We see a more complete **example**:

```
import astropy.units as u
from astropy.constants.si import c, G, M_sun, R_sun

wavelength = wavelength * u.AA


heliocentric = -23. * u.km/u.s
v_rad = -4.77 * u.km / u.s
R_MN_Lup = 0.9 * R_sun
M_MN_Lup = 0.6 * M_sun
vsini = 74.6 * u.km / u.s
period = 0.439 * u.day
```

Units and Constants

All numpy trigonometric functions expect the input in radian. We need to **convert** the angle manually using u.radian:

```
inclination = 45. * u.degree

incl = inclination.to(u.radian)
```

Units and Constants

Astronomy requires often the usage of logarithmic units such as **magnitudes**.

To create a logarithmic quantity:

```
import astropy.units as u, astropy.constants as c, numpy as np

u.Magnitude(-10.)
<Magnitude -10. mag>

u.Magnitude(10 * u.ct / u.s)
<Magnitude -2.5 mag(ct / s)>

u.Magnitude(-2.5, "mag(ct/s)")
<Magnitude -2.5 mag(ct / s)>

-2.5 * u.mag(u.ct / u.s)
<Magnitude -2.5 mag(ct / s)>
```

Units and Constants

Units that cancel out become a special unit called the **dimensionless unit**:

```
u.m / u.m
Unit(dimensionless)
```

To create a basic dimensionless quantity, we multiply a value by the unscaled dimensionless unit:

```
q = 1.0 * u.dimensionless_unscaled

q.unit
Unit(dimensionless)
```

More on working with units can be found in the documentation:
`https://docs.astropy.org/en/stable/units/`

with special astronomical applications such as photometric reduction:
`https://docs.astropy.org/en/stable/units/logarithmic_units.html`

Units and Constants

In addition to physical units as presented so far, we often deal with astronomical coordinate systems.

`astropy.units` is not capable of dealing with spherical geometry or sexagesimal (hours, min, sec) units: if you want to deal with celestial **coordinates**, see the package `astropy.coordinates`.

Astronomical Coordinate Systems

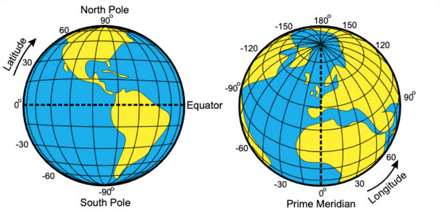
Astronomical coordinate systems are organized arrangements for specifying positions of satellites, planets, stars, galaxies, and other celestial objects relative to physical reference points available to a situated observer (e.g. the true horizon and north cardinal direction to an observer situated on the Earth's surface).

Coordinate systems in astronomy can be 3D (to specify an object's position in three-dimensional space) or 2D (its direction on a celestial sphere, if the object's distance is unknown or trivial).

Astronomical Coordinate Systems

Spherical coordinates, projected on the celestial sphere, are analogous to the geographic coordinate system used on the surface of Earth:

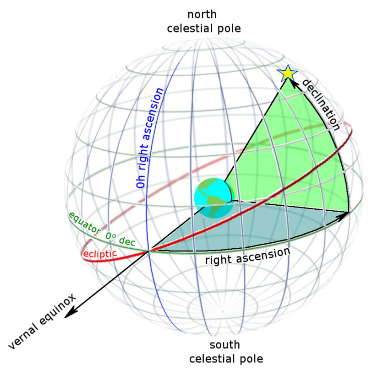
Geographic coordinate system:



Earth is shown covered in an imaginary grid of latitude lines (measured from 0° to 90° north and south of the equator) and longitudes lines (measured from 0° to 180° east and west of the prime meridian). You'll occasionally see the East and West longitude suffixes replaced by a negative sign for the western and a positive sign for the eastern hemisphere.

Astronomical Coordinate Systems

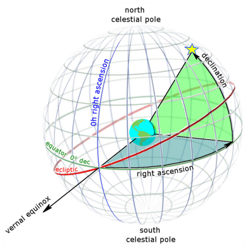
Equatorial system:



Declination (denoted as *Dec* or δ) is measured in degrees north and south of the celestial equator. Right ascension (denoted as *R.A.*, *RA* or α,), akin to longitude, is measured east from the equinox. The red circle is the Sun's apparent path around the sky, which defines the ecliptic.

Astronomical Coordinate Systems

Equatorial system:

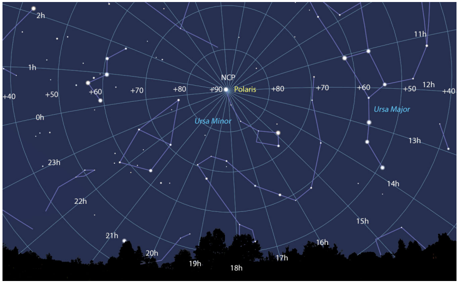


Anything north of the celestial equator has a northerly declination, marked with a positive sign.
Anything south of the equator has a negative declination written with a negative sign.

Example: Vega's declination is $+38^\circ\ 47'\ 1''$, while Alpha Centauri's is $-60^\circ\ 50'\ 2''$. One star is north of the celestial equator and the other south.

Astronomical Coordinate Systems

Equatorial system:

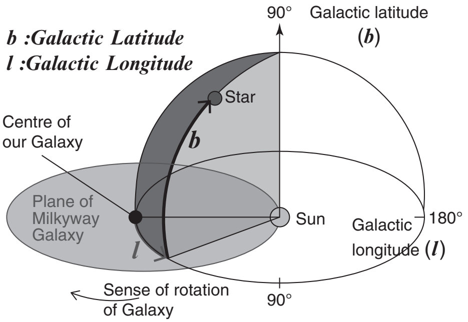


Right ascension can also be measured in **sidereal hours, minutes and seconds** instead of degrees, a result of measuring right ascensions by timing the passage of objects across the meridian as the Earth rotates. There are $360^\circ/24\mathrm{h} = 15^\circ$ in one hour of right ascension, and 24h of right ascension around the entire celestial equator.

This view (from the software Stellarium) shows the north celestial pole and polar regions. **Declinations** are labeled every 10°. The **hours of right ascension** are shown around the circle.

Astronomical Coordinate Systems

Galactic coordinate system:

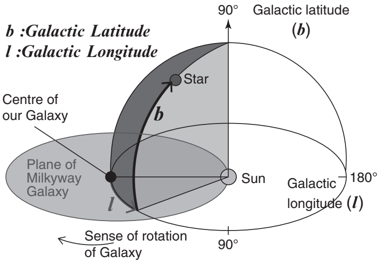


M. Inglis, Astronomy of the Milky Way, The Patrick Moore Practical

The galactic coordinate system is useful for describing an object's position with respect to the galaxy's center. For example, for an object with high galactic latitude, you might expect less obstruction by interstellar dust.

Astronomical Coordinate Systems

Galactic coordinate system:

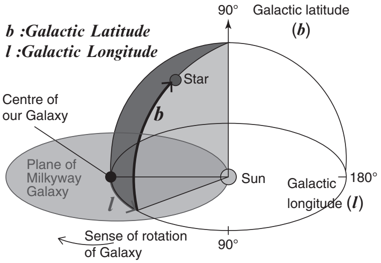


M. Inglis, Astronomy of the Milky Way, The Patrick Moore Practical

The first coordinate is the **galactic longitude** (*l* or *gl*), which is the angular separation of the object from the galaxy's "prime meridian", the great circle that passes through the Galactic center and the galactic poles. Galactic longitude is analogous to right ascension and is measured along the galactic equator in the same direction as right ascension. The zero-point of galactic longitude is in the direction of the Galactic Center.

Astronomical Coordinate Systems

Galactic coordinate system:



M. Inglis, Astronomy of the Milky Way, The Patrick Moore Practical

The second coordinate is the galactic latitude (b or gb) which denotes the angle an object makes with the galactic equator. The galactic equator runs through the center of the Milky Way's projection on the sky.
The galactic latitude is analogous to declination, but measures distance north or south of the galactic equator, attaining $+90^\circ$ at the north galactic pole (NGP) and -90° at the south galactic pole (SGP).

Astronomical Coordinate Systems

One can use the principles of spherical trigonometry as applied to triangles on the celestial sphere to derive equations for **transforming coordinates** from one coordinate system to in another. These equations generally rely on the spherical law of cosines.

Astronomical Coordinate Systems with `astropy`

`astropy` provides a framework to represent celestial coordinates and transform between them in its `astropy.coordinates` package.
Most of the common coordinate systems (ICRS, FK4, FK5, and Galactic, AltAz) are available. In addition, users can define their own systems if needed. Transformation of both individual scalar coordinates and arrays of coordinates are supported.

Coordinate objects are instantiated with a flexible and natural approach that supports both numeric angle values and (limited) string parsing:

The documentation can be found here:
`http://docs.astropy.org/en/stable/coordinates/index.html`

Astronomical Coordinate Systems with astropy

In the following examples, we will see how to use `astropy.coordinates`:

```
>>> from astropy.coordinates import SkyCoord
>>> from astropy import units as u
>>> SkyCoord(ra=10.68458, dec=41.26917, unit=(u.degree, u.degree))

<SkyCoord (ICRS): (ra, dec) in deg
    (10.68458, 41.26917)>

>>> SkyCoord('00h42m44.3s +41d16m9s')

<SkyCoord (ICRS): (ra, dec) in deg
    (10.68458333, 41.26916667)>
```

Astronomical Coordinate Systems with astropy

The individual components of a coordinate are `Angle` objects, and their values are accessed using special attributes:

```
>>> c = SkyCoord(ra=10.68458, dec=41.26917, unit=(u.degree, u.degree))
>>> c.ra

<<Longitude 10.68458 deg>

>>> c.ra.to('hourangle')

<Longitude 0.7123053333333335 hourangle>

>>> c.ra.hms

hms_tuple(h=0.0, m=42.0, s=44.299200000000525)

>>> c.dec

<<Latitude 41.26917 deg>
```

Astronomical Coordinate Systems with astropy

To **convert** to some other coordinate system, the easiest method is to use attribute-style access with short names for the built-in systems:

```
>>> c.galactic

<SkyCoord (Galactic): (l, b) in deg
    (121.17424181, -21.57288557)>
```

The `astropy.coordinates` subpackage also provides a quick way to get coordinates for named objects (with an internet connection). All coordinate classes have a special class method, `from_name()`, that accepts a string and queries the service `Sesame` to retrieve coordinates for that object:

```
>>> c_eq = SkyCoord.from_name("M16")
>>> c_eq

<SkyCoord (ICRS): (ra, dec) in deg
    (274.7, -13.8067)>
```

Astronomical Coordinate Systems with `astropy`

So far, we have used `SkyCoord` to represent angular on-sky positions (i.e., RA and Dec).

In many cases, however, it is useful to **include distance information** with the sky coordinates of a source, thereby specifying the full 3D position of an object.

Astronomical Coordinate Systems with astropy

So far, we have used SkyCoord to represent angular on-sky positions (i.e., RA and Dec).

In many cases, however, it is useful to **include distance information** with the sky coordinates of a source, thereby specifying the full 3D position of an object.

To pass in distance information, SkyCoord accepts the keyword argument distance. So, if we knew that the distance to NGC 188 is 1.96 kpc, we could also pass in a distance (as a Quantity object) using this argument:

```
ngc188_center_3d = SkyCoord(12.11*u.deg, 85.26*u.deg,
                            distance=1.96*u.kpc)
```

Astronomical Coordinate Systems with astropy

example: Querying the Gaia Archive to retrieve coordinates of possible star cluster member stars

We use a SkyCoord object for the center of NGC 188. We thento select nearby sources from the Gaia Data Release 2 catalog using the astroquery.gaia package. For this, an internet connection is required.

```
ngc188_center_3d = SkyCoord(12.11*u.deg, 85.26*u.deg,
                            distance=1.96*u.kpc)

job = Gaia.cone_search_async(ngc188_center, radius=0.5*u.deg)
ngc188_table = job.get_results()

# only keep stars brighter than G=19 magnitude
ngc188_table =
     ngc188_table[ngc188_table['phot_g_mean_mag'] < 19*u.mag]
```

Astronomical Coordinate Systems with astropy

With the table of Gaia data we retrieved above for stars around NGC 188, ngc188_table, we also have parallax measurements for each star. For a precisely-measured parallax ω in arcsec, the distance d to a star can be obtained approximately as $d \sim 1/\omega$ using units of parsec, as one parsec is equal to the parallax of one arcsecond.

This only really works if the parallax error is small relative to the parallax, so if we want to use these parallaxes to get distances we first have to filter out stars that have low signal-to-noise parallaxes:

```
parallax_snr =
    ngc188_table['parallax'] / ngc188_table['parallax_error']
ngc188_table_3d = ngc188_table[parallax_snr > 10]

print(len(ngc188_table_3d))
```

Astronomical Coordinate Systems with astropy

With the table of Gaia data we retrieved above for stars around NGC 188, `ngc188_table`, we also have parallax measurements for each star. For a precisely-measured parallax ω in arcsec, the distance d to a star can be obtained approximately as $d \sim 1/\omega$ using units of parsec, as one parsec is equal to the parallax of one arcsecond.

This only really works if the parallax error is small relative to the parallax, so if we want to use these parallaxes to get distances we first have to filter out stars that have low signal-to-noise parallaxes:

```
parallax_snr =
     ngc188_table['parallax'] / ngc188_table['parallax_error']
ngc188_table_3d = ngc188_table[parallax_snr > 10]

print(len(ngc188_table_3d))
```

we get

```
2045
```

Astronomical Coordinate Systems with `astropy`

The above selection on `parallax_snr` keeps stars that have a ∼10-sigma parallax measurement, but this is an arbitrary selection threshold that you may want to adjust to your own use cases. Here, this selection removed over half of the stars in our original table.

For the remaining stars we can be confident that converting the parallax measurements to distances is mostly safe.

Astronomical Coordinate Systems with astropy

The above selection on `parallax_snr` keeps stars that have a ∼10-sigma parallax measurement, but this is an arbitrary selection threshold that you may want to adjust to your own use cases. Here, this selection removed over half of the stars in our original table.

For the remaining stars we can be confident that converting the parallax measurements to distances is mostly safe.

The default way of passing a distance to a SkyCoord object, as above, is to pass in a `Quantity`. However, `astropy.coordinates` also provides a specialized object, `Distance`, for handling common transformations of different distance representations. This class supports passing in a parallax value:

```
Distance(parallax=1*u.mas)
```

Astronomical Coordinate Systems with astropy

The catalog of stars we queried from Gaia contains parallax information in milliarcsecond units, so we can create a Distance object directly from these values:

```
gaia_dist =
     Distance(parallax=ngc188_table_3d['parallax'].filled(np.nan))
```

We can then create a `SkyCoord` object representing the 3D positions of all the stars by initializing `SkyCoord` with the `distance` object:

```
ngc188_coords_3d = SkyCoord(ra=ngc188_table_3d['ra'],
                            dec=ngc188_table_3d['dec'],
                            distance=gaia_dist)
print(ngc188_coords_3d)
```

Astronomical Coordinate Systems with astropy

This gives:

```
<SkyCoord (ICRS): (ra, dec, distance) in (deg, deg, pc)
    [(11.77583526, 85.24193176, 1732.33856712),
     (11.79609228, 85.24947782, 1807.88146705),
     (11.7247587 , 85.24364992, 1967.48335718), ...,
     (13.6808148 , 84.78031087, 1132.77844818),
     ( 6.2852781 , 85.08022389,  158.78970106),
     ( 7.46880607, 84.9195982 , 1767.89065721)]>
```

Astronomical Coordinate Systems with astropy

Often we want to create plots. We now use `matplotlib` to plot the sky positions of all of these sources, colored by distance to emphasize the cluster stars:

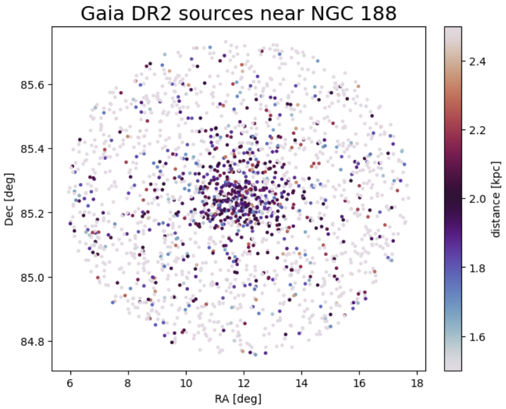
```
fig, ax = plt.subplots(figsize=(6.5, 5.2),
                       constrained_layout=True)
cs = ax.scatter(ngc188_coords_3d.ra.degree,
                ngc188_coords_3d.dec.degree,
                c=ngc188_coords_3d.distance.kpc,
                s=5, vmin=1.5, vmax=2.5, cmap='twilight')
cb = fig.colorbar(cs)
cb.set_label(f'distance [{u.kpc:latex_inline}]')

ax.set_xlabel('RA [deg]')
ax.set_ylabel('Dec [deg]')

ax.set_title('Gaia DR2 sources near NGC 188', fontsize=18)
```

Astronomical Coordinate Systems with astropy

We get the following figure:



Astronomical Coordinate Systems with astropy

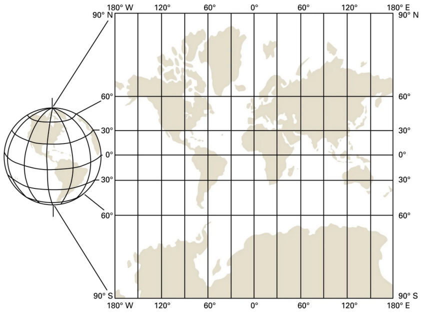
Finally, that we have 3D position information for both the cluster center, and for the stars we queried from Gaia, we can compute the 3D separation (distance) between all of the Gaia sources and the cluster center:

```
sep3d = ngc188_coords_3d.separation_3d(ngc188_center_3d)
```

Map Projections

How to force a **spherical surface** (i.e.: the Earth's surface, the apparent sphere of the sky) onto a flat map is an old problem in cartography.

Consider how the cylindrical projection into Cartesian coordinates greatly inflates the area of the poles (e.g., Antarctica) in the map:



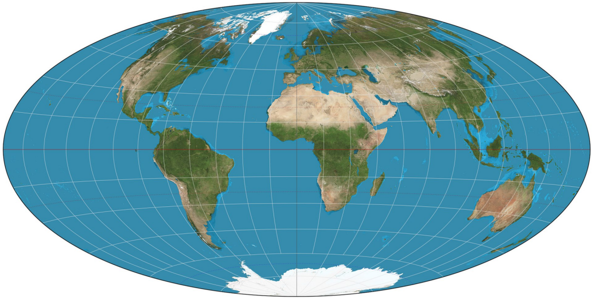
The Mercator projection was presented by Gerardus Mercator in 1569. Credit: https://www.britannica.com/science/Mercator-projection

Map Projections

It is possible to create **Equal Area Projections**: mappings in which spherical surfaces are equal-area when projected onto a flat surface.

A common solution , also in astronomy, is the Hammer-Aitoff projection. In the example below, note how much smaller Antartica is now.

Another commonly used equal-area projection is the Mollweide projection.



The world in Hammer-Aitoff projection. Credit: Strebe

Map Projections in Python

Map Projections such as the Hammer-Aitoff are available in `matplotlib`. For getting our data in the right angular format, we use `astropy.coordinates.Angle`.
The general set of commands is the following:

```python
fig = plt.figure()

#here 111 means a subplot 1 of a 1 times 1 grid of plots
ax = fig.add_subplot(111, projection="aitoff")

#ra,dec must be given in radians with -pi < ra < pi
ra = coord.Angle(data['RA'].filled(np.nan)*u.degree)
ra = ra.wrap_at(180*u.degree)
dec = coord.Angle(data['Dec'].filled(np.nan)*u.degree)

ax.scatter(ra.radian, dec.radian)
fig.show()
```

Astronomical Time Formats

solar time, siderial time:

An Earth day can be measured in different ways:

- The time it takes for a complete rotation of Earth around its axis.
- Measure the time it takes for the Sun to appear in the same meridian in the sky. This interval is known as the **solar day**.
- Measure the time it takes for a distant star to appear in the same meridian in the sky. This interval is known as the **sidereal day**.

Astronomical Time Formats

solar time, siderial time:

An Earth day can be measured in different ways:

- The time it takes for a complete rotation of Earth around its axis.
- Measure the time it takes for the Sun to appear in the same meridian in the sky. This interval is known as the **solar day**.
- Measure the time it takes for a distant star to appear in the same meridian in the sky. This interval is known as the **sidereal day**.

The first definition of a day is equal to the sidereal day, as we measure the angular rotation relative to distant quasars.

A complete rotation of Earth around its axis (= a sideral day) takes 23 hours, 56 minutes and 4.1 seconds.

Astronomical Time Formats

solar time, siderial time:

Earth moves a little less than a degree around the Sun during the time it takes for one full axial rotation.
For the Sun to appear on the same meridian in the sky again after a full axial rotation, the Earth has to rotate one extra degree to bring the Sun into the same apparent meridian in the sky. This is why the solar day is longer than the sidereal day by about 4 minutes.

Astronomical Time Formats

solar time, siderial time:

Earth moves a little less than a degree around the Sun during the time it takes for one full axial rotation.
For the Sun to appear on the same meridian in the sky again after a full axial rotation, the Earth has to rotate one extra degree to bring the Sun into the same apparent meridian in the sky. This is why the solar day is longer than the sidereal day by about 4 minutes.

⇨ A star will rise 4 minutes earlier each night: tomorrow, you must observe 4 minutes earlier for the same star to be on your meridian than today (a given RA is on your meridian 4 minutes earlier each day).

Astronomical Time Formats

(Modified) Julian date:

Given the different time systems, leap years etc. it is useful to have a calendar with which to **express exact times of observations** (referred to as **epochs**).

In astronomy, times based on the Julian Date are used.

Julian Date (JD) is a count in days from 0 at noon on January the 1st in the year -4712 (4713 BC).

The integral part gives the Julian day number. The fractional part gives the time of day since noon UT as a decimal fraction of one day or fractional day, with 0.5 representing midnight UT.

Astronomical Time Formats

(Modified) Julian date:

Modified Julian Date (MJD) is a count in days from 0 midnight on November 17 in the year 1858:
MJD = JD - 2400000.5

Reduced Julian Date (RJD):
RJD = JD - 2400000

MJD and RJD are commonly used to reduce the number of digits - less digits to save, easier to plot.

Astronomical Time Formats in Python

The `astropy.time` package provides methods for manipulating times and dates. Specific emphasis is placed on supporting time scales (e.g., UTC, TAI, UT1, TDB) and time representations (e.g., JD, MJD, ISO 8601) that are used in astronomy.

All time manipulations and arithmetic operations are done internally using 64-bit floats to represent time.

The usual way to use `astropy.time` is by creating a `Time` object to which we supply one or more input time values as well as the time format and time scale of those values.

The input time(s) can either be a single scalar like `"2010-01-01 00:00:00"` or a list or a `numpy` array of values as shown below. In general, any output values have the same shape (scalar or array) as the input.

Astronomical Time Formats in Python

To create a Time object:

```python
import numpy as np
from astropy.time import Time

times = ['1999-01-01T00:00:00.123456789', '2010-01-01T00:00:00']

t = Time(times, format='isot', scale='utc')

print(t)
print(t[1])
```

```
<Time object: scale='utc' format='isot'
value=['1999-01-01T00:00:00.123' '2010-01-01T00:00:00.000']>

<Time object: scale='utc' format='isot' value=2010-01-01T00:00:00.000>
```

Astronomical Time Formats in Python

Now we can get the representation of these times in the JD and MJD formats by requesting the corresponding Time attributes:

```
print(t.jd)
print(t.mjd)
```

```
array([2451179.50000143, 2455197.5       ])

array([51179.00000143, 55197.        ])
```

more on this can be found in the documentation:
`https://docs.astropy.org/en/stable/time/`

Summary

We have seen many ways on how to work with astronomical data, such as accessing and transforming especially time series data.

In this week's **tutorial session** we will continue with a more complete example of processing astronomical data in a Jupyter notebook that summarizes what we have learnt during this course.

Outlook

In the next lecture, we will see how we can successfully structure a small software project:

How we can **debug and test** our code, write **documentation** and use **github**.