Python (Semester 2 2024)

# Astronomical Packages

**Nina Hernitschek**
Centro de Astronomía CITEVA
Universidad de Antofagasta

October 30, 2024

# Motivation



We have seen so far that Python provides a lot 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**. In Astroinformatica I, we have seen some astronomical packages, especially `astropy`. We will briefly review `astropy` before continuing with more specialized astronomical packages for Python.

# Units and Constants

Despite calculations are often done without keeping the units, it makes sense to keep track of the units to avoid e.g. mixing up values.

# Units and Constants



Despite calculations are often done without keeping the units, it makes sense to keep track of the units to avoid e.g. mixing up values.
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`

# 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



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).
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 astronomical 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 $\omega$ 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 $\omega$ 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 $\sim$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)
```

# Map Projections in Python



The `matplotlib` package contains functionality to create map projections such as the Hammer-Aitoff projection. For getting our data in the right angular format, we use `astropy.coordinates.Angle`.
The general set of commands is the following:

```
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



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.

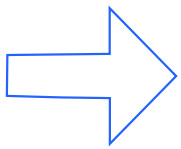
# Astronomical Time Formats



## (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:

```
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/`

# Processing Photometry with `photutils`




Photutils is an affiliated open-source package of Astropy providing tools for detecting and performing photometry of astronomical sources.

Its functionality includes e.g.:

- performing aperture photometry
- performing PSF-fitting photometry
- detecting and extracting point-like sources (e.g., stars)
- detecting and extracting extended sources using image segmentation
- estimating the background in astronomical images
- estimating morphological parameters of detected sources
- estimating the limiting depths of images

The documentation is available at `https://photutils.readthedocs.io/`

# Source Detection with `photutils.detection`

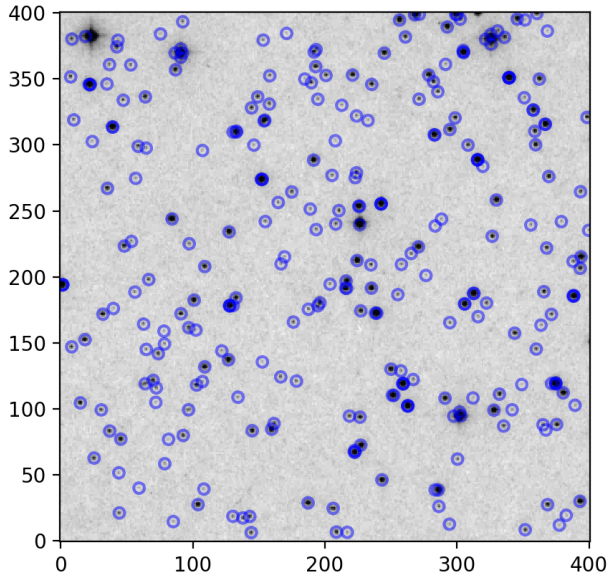


Before we can perform photometry or morphological measurements, astronomical sources have to be identified in astronomical images.

When we work with astronomical catalog data, this step has already been performed. Also, when we take targeted observations as part of an observing program, we already know the (few) sources we deal with.

However, if processing raw imaging data containing many sources this step is essential.

`photutils.detection` provides several tools designed specifically to detect point-like (stellar) sources in an astronomical image.
In addition, a function to identify local peaks in an image that are above a specified threshold value is provided.



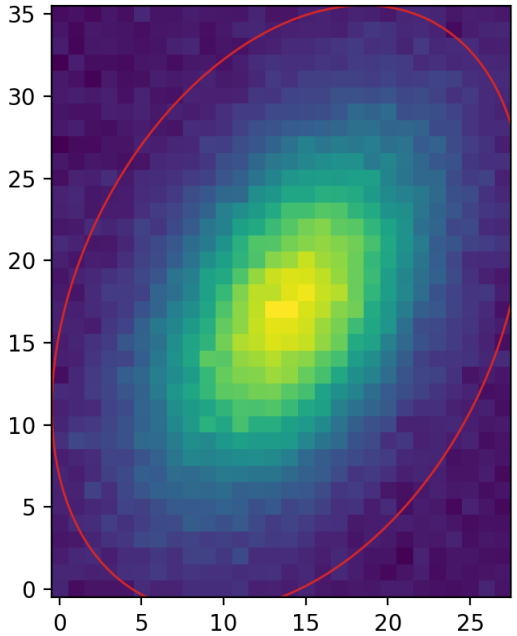

credit:
`https://photutils.readthedocs.io/`

# Morphology Properties with `photutils.morphology`



The `data_properties()` function can be used to calculate the morphological properties of a single source in a cutout image. `data_properties` returns a `SourceCatalog` object. Please see `SourceCatalog` for the list of the many properties that are calculated. Even more properties are likely to be added in the future.

In case of a segmentation image, the `SourceCatalog` class can be used to calculate the properties for all (or a specified subset) of the segmented sources.




credit:
`https://photutils.readthedocs.io/`

# Processing Spectra with `specutils`




Specutils is a Python package for representing, loading, manipulating, and analyzing astronomical spectroscopic data. The generic data containers and accompanying modules provide a toolbox that the astronomical community can use to build more domain-specific packages.

The documentation is available at `https://specutils.readthedocs.io/`

# Processing Spectra with specutils



**example:** emission line galaxy spectrum from SDSS

We begin with some imports:

```
from astropy.io import fits
from astropy import units as u
import numpy as np
from matplotlib import pyplot as plt
from astropy.visualization import quantity_support
quantity_support()  # for getting units on the axes below
```

Now we load the dataset from its source:

```
filename = 'https://data.sdss.org/sas/dr16/sdss/spectro/'
           +'redux/26/spectra/1323/spec-1323-52797-0012.fits'

# The spectrum is in the second HDU of this file.
with fits.open(filename) as f:
    specdata = f[1].data
```

# Processing Spectra with specutils



We then re-format this dataset into astropy quantities, and create a `Spectrum1D` object:

```
from specutils import Spectrum1D

lamb = 10**specdata['loglam'] * u.AA
flux = specdata['flux'] * 10**-17 * u.Unit('erg cm-2 s-1 AA-1')
spec = Spectrum1D(spectral_axis=lamb, flux=flux)
```

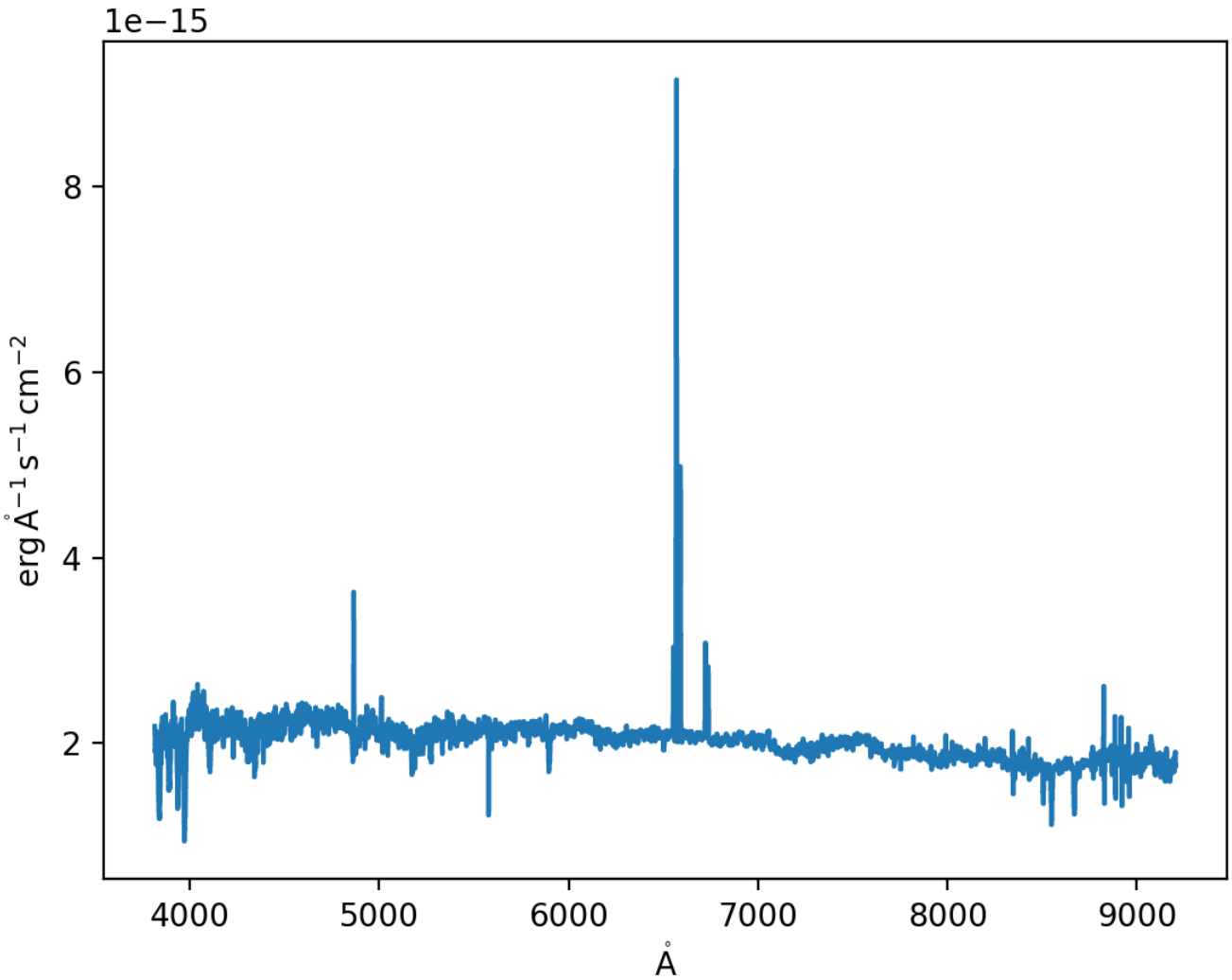
We plot the spectrum:

```
f, ax = plt.subplots()

ax.step(spec.spectral_axis, spec.flux)
```

# Processing Spectra with `specutils`



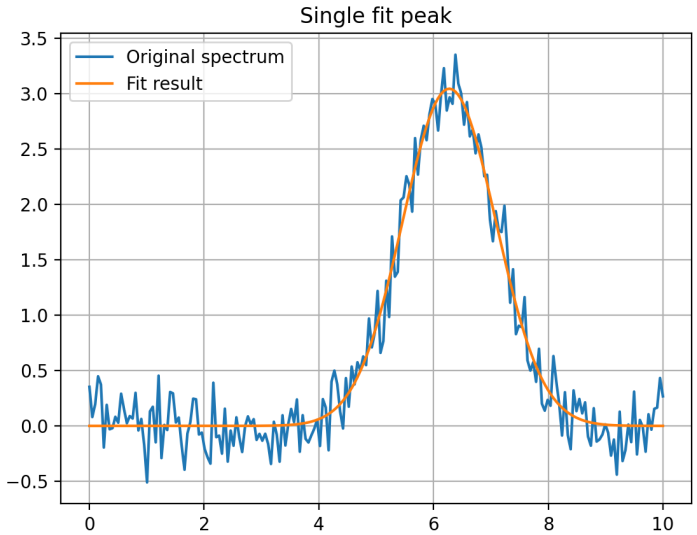


# Processing Spectra with `specutils`



One of the primary tasks in spectroscopic analysis is fitting models, such as line and continuum fitting. `specutils` provides conveniences that aim to leverage the general fitting framework of `astropy.modeling` to spectral-specific tasks.

At a high level, this fitting takes the `Spectrum1D` object and a list of Model objects that have initial guesses for each of the parameters. these are used to create a compound model created from the model initial guesses. This model is then actually fit to the spectrum's flux, yielding a single composite model result (which can be split back into its components if desired).



# Overview: Other Packages



A list of astronomical packages for Python affiliated with `astropy` can be found under the following URL:
`https://www.astropy.org/affiliated/`

Generally, for packages affiliated with `astropy` one can expect that they are compatible with `astropy`, and have the same high quality of code as well as documentation.

# Overview: Other Packages



A list of astronomical packages for Python affiliated with `astropy` can be found under the following URL:
`https://www.astropy.org/affiliated/`

Generally, for packages affiliated with `astropy` one can expect that they are compatible with `astropy`, and have the same high quality of code as well as documentation.

If the desired functionality cannot be found in any of those packages, a good strategy is to look up papers implementing a similar strategy and looking up what the authors are using. Often nowadays code from a paper can be found on a github repository mentioned in said paper.

# Summary



We have seen many ways on how to work with astronomical data, including photometry and spectra.

Summarizing what you have learned in the previous classes, with this you should be prepared to solve astronomical research questions with code you write in Python, as well as ensure the correctness and efficincy of your code.