2. Coordinates and Units

Print to PDF

Contents

Outline
Working with Units
Exercise
Selecting a Region
Exercise
Getting GD-1 Data
Transforming coordinates
Exercise
Selecting a rectangle
Defining a polygon
Assembling the query
Saving results
Summary
Best practices

In the previous lesson, we wrote ADQL queries and used them to select and download data from the Gaia server.

In this lesson, we'll pick up where we left off and write a query to select stars from a particular region of the sky.

Outline

We'll start with an example that does a "cone search"; that is, it selects stars that appear in a circular region of the sky.

Then, to select stars in the vicinity of GD-1, we'll:

- Use `Quantity` objects to represent measurements with units.
- Use Astropy to convert coordinates from one frame to another.
- Use the ADQL keywords `POLYGON`, `CONTAINS`, and `POINT` to select stars that fall within a polygonal region.
- Submit a query and download the results.
- Store the results in a FITS file.

After completing this lesson, you should be able to

- Use Python string formatting to compose more complex ADQL queries.
- Work with coordinates and other quantities that have units.
- Download the results of a query and store them in a file.

Working with Units

The measurements we will work with are physical quantities, which means that they have two parts, a value and a unit. For example, the coordinate 30° has value 30 and its units are degrees.

Until recently, most scientific computation was done with values only; units were left out of the program altogether, often with catastrophic results.

Astropy provides tools for including units explicitly in computations, which makes it possible to detect errors before they cause disasters.

To use Astropy units, we import them like this:

```
import astropy.units as u
```

`u` is an object that contains most common units and all SI units.

You can use `dir` to list them, but you should also read the documentation.

```
dir(u)
```

```
['A',
 'AA',
 'AB',
 'ABflux',
 'ABmag',
 'AU',
 'Angstrom',
 'B',
 'Ba',
 'Barye',
 'Bi',
 'Biot',
 'Bol',
 'Bq',
 'C',
 'Celsius',
 'Ci',
 'CompositeUnit',
 'D',
 'Da',
 'Dalton',
 'Debye',
 'Decibel',
 'DecibelUnit',
 'Dex',
 'DexUnit',
 'EA',
 'EAU',
 'EB',
 'EBa',
 'EC',
 'ED',
 'EF',
 'EG',
 'EGal',
 'EH',
 'EHz',
 'EJ',
 'EJy',
 'EK',
 'EL',
 'EN',
 'EOhm',
 'EP',
 'EPa',
 'ER',
 'ERy',
 'ES',
 'ESt',
 'ET',
 'EV',
 'EW',
 'EWb',
 'Ea',
 'Eadu',
 'Earcmin',
 'Earcsec',
 'Eau',
 'Eb',
 'Ebarn',
 'Ebeam',
 'Ebin',
 'Ebit',
 'Ebyte',
 'Ecd',
 'Echan',
 'Ecount',
 'Ect',
 'Ed',
 'Edeg',
 'Edyn',
 'EeV',
 'Eerg',
 'Eg',
 'Eh',
 'EiB',
 'Eib',
 'Eibit',
 'Eibyte',
 'Ek',
 'El',
 'Elm',
 'Elx',
 'Elyr',
 'Em',
 'Emag',
 'Emin',
```

```
'Emol',
'Eohm',
'Epc',
'Eph',
'Ephoton',
'Epix',
'Epixel',
'Erad',
'Es',
'Esr',
'Eu',
'Evox',
'Evoxel',
'Eyr',
'F',
'Farad',
'Fr',
'Franklin',
'FunctionQuantity',
'FunctionUnitBase',
'G',
'GA',
'GAU',
'GB',
'GBa',
'GC',
'GD',
'GF',
'GG',
'GGal',
'GH',
'GHz',
'GJ',
'GJy',
'GK',
'GL',
'GN',
'GOhm',
'GP',
'GPa',
'GR',
'GRy',
'GS',
'GSt',
'GT',
'GV',
'GW',
'GWb',
'Ga',
'Gadu',
'Gal',
'Garcmin',
'Garcsec',
'Gau',
'Gauss',
'Gb',
'Gbarn',
'Gbeam',
'Gbin',
'Gbit',
'Gbyte',
'Gcd',
'Gchan',
'Gcount',
'Gct',
'Gd',
'Gdeg',
'Gdyn',
'GeV',
'Gerg',
'Gg',
'Gh',
'GiB',
'Gib',
'Gibit',
'Gibyte',
'Gk',
'Gl',
'Glm',
'Glx',
'Glyr',
'Gm',
'Gmag',
'Gmin',
'Gmol',
'Gohm',
'Gpc',
'Gph',
```

```
'Gphoton',
'Gpix',
'Gpixel',
'Grad',
'Gs',
'Gsr',
'Gu',
'Gvox',
'Gvoxel',
'Gyr',
'H',
'Henry',
'Hertz',
'Hz',
'IrreducibleUnit',
'J',
'Jansky',
'Joule',
'Jy',
'K',
'Kayser',
'Kelvin',
'KiB',
'Kib',
'Kibit',
'Kibyte',
'L',
'L_bol',
'L_sun',
'LogQuantity',
'LogUnit',
'Lsun',
'MA',
'MAU',
'MB',
'MBa',
'MC',
'MD',
'MF',
'MG',
'MGal',
'MH',
'MHz',
'MJ',
'MJy',
'MK',
'ML',
'MN',
'MOhm',
'MP',
'MPa',
'MR',
'MRy',
'MS',
'MSt',
'MT',
'MV',
'MW',
'MWb',
'M_bol',
'M_e',
'M_earth',
'M_jup',
'M_jupiter',
'M_p',
'M_sun',
'Ma',
'Madu',
'MagUnit',
'Magnitude',
'Marcmin',
'Marcsec',
'Mau',
'Mb',
'Mbarn',
'Mbeam',
'Mbin',
'Mbit',
'Mbyte',
'Mcd',
'Mchan',
'Mcount',
'Mct',
'Md',
'Mdeg',
'Mdyn',
'MeV',
'Mearth',
```

```
'Merg',
'Mg',
'Mh',
'MiB',
'Mib',
'Mibit',
'Mibyte',
'Mjup',
'Mjupiter',
'Mk',
'Ml',
'Mlm',
'Mlx',
'Mlyr',
'Mm',
'Mmag',
'Mmin',
'Mmol',
'Mohm',
'Mpc',
'Mph',
'Mphoton',
'Mpix',
'Mpixel',
'Mrad',
'Ms',
'Msr',
'Msun',
'Mu',
'Mvox',
'Mvoxel',
'Myr',
'N',
'NamedUnit',
'Newton',
'Ohm',
'P',
'PA',
'PAU',
'PB',
'PBa',
'PC',
'PD',
'PF',
'PG',
'PGal',
'PH',
'PHz',
'PJ',
'PJy',
'PK',
'PL',
'PN',
'POhm',
'PP',
'PPa',
'PR',
'PRy',
'PS',
'PSt',
'PT',
'PV',
'PW',
'PWb',
'Pa',
'Padu',
'Parcmin',
'Parcsec',
'Pascal',
'Pau',
'Pb',
'Pbarn',
'Pbeam',
'Pbin',
'Pbit',
'Pbyte',
'Pcd',
'Pchan',
'Pcount',
'Pct',
'Pd',
'Pdeg',
'Pdyn',
'PeV',
'Perg',
'Pg',
'Ph',
'PiB',
```

```
'Pib',
'Pibit',
'Pibyte',
'Pk',
'Pl',
'Plm',
'Plx',
'Plyr',
'Pm',
'Pmag',
'Pmin',
'Pmol',
'Pohm',
'Ppc',
'Pph',
'Pphoton',
'Ppix',
'Ppixel',
'Prad',
'PrefixUnit',
'Ps',
'Psr',
'Pu',
'Pvox',
'Pvoxel',
'Pyr',
'Quantity',
'QuantityInfo',
'QuantityInfoBase',
'R',
'R_earth',
'R_jup',
'R_jupiter',
'R_sun',
'Rayleigh',
'Rearth',
'Rjup',
'Rjupiter',
'Rsun',
'Ry',
'S',
'ST',
'STflux',
'STmag',
'Siemens',
'SpecificTypeQuantity',
'St',
'Sun',
'T',
'TA',
'TAU',
'TB',
'TBa',
'TC',
'TD',
'TF',
'TG',
'TGal',
'TH',
'THz',
'TJ',
'TJy',
'TK',
'TL',
'TN',
'TOhm',
'TP',
'TPa',
'TR',
'TRy',
'TS',
'TSt',
'TT',
'TV',
'TW',
'TWb',
'Ta',
'Tadu',
'Tarcmin',
'Tarcsec',
'Tau',
'Tb',
'Tbarn',
'Tbeam',
'Tbin',
'Tbit',
'Tbyte',
'Tcd',
```

```
'Tchan',
'Tcount',
'Tct',
'Td',
'Tdeg',
'Tdyn',
'TeV',
'Terg',
'Tesla',
'Tg',
'Th',
'TiB',
'Tib',
'Tibit',
'Tibyte',
'Tk',
'Tl',
'Tlm',
'Tlx',
'Tlyr',
'Tm',
'Tmag',
'Tmin',
'Tmol',
'Tohm',
'Torr',
'Tpc',
'Tph',
'Tphoton',
'Tpix',
'Tpixel',
'Trad',
'Ts',
'Tsr',
'Tu',
'Tvox',
'Tvoxel',
'Tyr',
'Unit',
'UnitBase',
'UnitConversionError',
'UnitTypeError',
'UnitsError',
'UnitsWarning',
'UnrecognizedUnit',
'V',
'Volt',
'W',
'Watt',
'Wb',
'Weber',
'YA',
'YAU',
'YB',
'YBa',
'YC',
'YD',
'YF',
'YG',
'YGal',
'YH',
'YHz',
'YJ',
'YJy',
'YK',
'YL',
'YN',
'YOhm',
'YP',
'YPa',
'YR',
'YRy',
'YS',
'YSt',
'YT',
'YV',
'YW',
'YWb',
'Ya',
'Yadu',
'Yarcmin',
'Yarcsec',
'Yau',
'Yb',
'Ybarn',
'Ybeam',
'Ybin',
'Ybit',
```

```
 'Ybyte',
 'Ycd',
 'Ychan',
 'Ycount',
 'Yct',
 'Yd',
 'Ydeg',
 'Ydyn',
 'YeV',
 'Yerg',
 'Yg',
 'Yh',
 'Yk',
 'Yl',
 'Ylm',
 'Ylx',
 'Ylyr',
 'Ym',
 'Ymag',
 'Ymin',
 'Ymol',
 'Yohm',
 'Ypc',
 'Yph',
 'Yphoton',
 'Ypix',
 'Ypixel',
 'Yrad',
 'Ys',
 'Ysr',
 'Yu',
 'Yvox',
 'Yvoxel',
 'Yyr',
 'ZA',
 'ZAU',
 'ZB',
 'ZBa',
 'ZC',
 'ZD',
 'ZF',
 'ZG',
 'ZGal',
 'ZH',
 'ZHz',
 'ZJ',
 'ZJy',
 'ZK',
 'ZL',
 'ZN',
 'ZOhm',
 'ZP',
 'ZPa',
 'ZR',
 'ZRy',
 'ZS',
 'ZSt',
 'ZT',
 'ZV',
 'ZW',
 'ZWb',
 'Za',
 'Zadu',
 'Zarcmin',
 'Zarcsec',
 'Zau',
 'Zb',
 'Zbarn',
 'Zbeam',
 'Zbin',
 'Zbit',
 'Zbyte',
 'Zcd',
 'Zchan',
 'Zcount',
 'Zct',
 'Zd',
 'Zdeg',
 'Zdyn',
 'ZeV',
 'Zerg',
 'Zg',
 'Zh',
 'Zk',
 'Zl',
 'Zlm',
 'Zlx',
 'Zlyr',
```

```
'Zm',
'Zmag',
'Zmin',
'Zmol',
'Zohm',
'Zpc',
'Zph',
'Zphoton',
'Zpix',
'Zpixel',
'Zrad',
'Zs',
'Zsr',
'Zu',
'Zvox',
'Zvoxel',
'Zyr',
'__builtins__',
'__cached__',
'__doc__',
'__file__',
'__loader__',
'__name__',
'__package__',
'__path__',
'__spec__',
'a',
'aA',
'aAU',
'aB',
'aBa',
'aC',
'aD',
'aF',
'aG',
'aGal',
'aH',
'aHz',
'aJ',
'aJy',
'aK',
'aL',
'aN',
'aOhm',
'aP',
'aPa',
'aR',
'aRy',
'aS',
'aSt',
'aT',
'aV',
'aW',
'aWb',
'aa',
'aadu',
'aarcmin',
'aarcsec',
'aau',
'ab',
'abA',
'abC',
'abampere',
'abarn',
'abcoulomb',
'abeam',
'abin',
'abit',
'abyte',
'acd',
'achan',
'acount',
'act',
'ad',
'add_enabled_equivalencies',
'add_enabled_units',
'adeg',
'adu',
'adyn',
'aeV',
'aerg',
'ag',
'ah',
'ak',
'al',
'allclose',
'alm',
'alx',
```

```
'alyr',
'am',
'amag',
'amin',
'amol',
'amp',
'ampere',
'angstrom',
'annum',
'aohm',
'apc',
'aph',
'aphoton',
'apix',
'apixel',
'arad',
'arcmin',
'arcminute',
'arcsec',
'arcsecond',
'asr',
'astronomical_unit',
'astrophys',
'attoBarye',
'attoDa',
'attoDalton',
'attoDebye',
'attoFarad',
'attoGauss',
'attoHenry',
'attoHertz',
'attoJansky',
'attoJoule',
'attoKayser',
'attoKelvin',
'attoNewton',
'attoOhm',
'attoPascal',
'attoRayleigh',
'attoSiemens',
'attoTesla',
'attoVolt',
'attoWatt',
'attoWeber',
'attoamp',
'attoampere',
'attoannum',
'attoarcminute',
'attoarcsecond',
'attoastronomical_unit',
'attobarn',
'attobarye',
'attobit',
'attobyte',
'attocandela',
'attocoulomb',
'attocount',
'attoday',
'attodebye',
'attodegree',
'attodyne',
'attoelectronvolt',
'attofarad',
'attogal',
'attogauss',
'attogram',
'attohenry',
'attohertz',
'attohour',
'attohr',
'attojansky',
'attojoule',
'attokayser',
'attolightyear',
'attoliter',
'attolumen',
'attolux',
'attometer',
'attominute',
'attomole',
'attonewton',
'attoparsec',
'attopascal',
'attophoton',
'attopixel',
'attopoise',
'attoradian',
'attorayleigh',
```

```
'attorydberg',
'attosecond',
'attosiemens',
'attosteradian',
'attostokes',
'attotesla',
'attovolt',
'attovoxel',
'attowatt',
'attoweber',
'attoyear',
'au',
'avox',
'avoxel',
'ayr',
'b',
'bar',
'barn',
'barye',
'beam',
'beam_angular_area',
'becquerel',
'bin',
'binary_prefixes',
'bit',
'bol',
'brightness_temperature',
'byte',
'cA',
'cAU',
'cB',
'cBa',
'cC',
'cD',
'cF',
'cG',
'cGal',
'cH',
'cHz',
'cJ',
'cJy',
'cK',
'cL',
'cN',
'cOhm',
'cP',
'cPa',
'cR',
'cRy',
'cS',
'cSt',
'cT',
'cV',
'cW',
'cWb',
'ca',
'cadu',
'candela',
'carcmin',
'carcsec',
'cau',
'cb',
'cbarn',
'cbeam',
'cbin',
'cbit',
'cbyte',
'ccd',
'cchan',
'ccount',
'cct',
'cd',
'cdeg',
'cdyn',
'ceV',
'centiBarye',
'centiDa',
'centiDalton',
'centiDebye',
'centiFarad',
'centiGauss',
'centiHenry',
'centiHertz',
'centiJansky',
'centiJoule',
'centiKayser',
'centiKelvin',
'centiNewton',
```

```
'centiOhm',
'centiPascal',
'centiRayleigh',
'centiSiemens',
'centiTesla',
'centiVolt',
'centiWatt',
'centiWeber',
'centiamp',
'centiampere',
'centiannum',
'centiarcminute',
'centiarcsecond',
'centiastronomical_unit',
'centibarn',
'centibarye',
'centibit',
'centibyte',
'centicandela',
'centicoulomb',
'centicount',
'centiday',
'centidebye',
'centidegree',
'centidyne',
'centielectronvolt',
'centifarad',
'centigal',
'centigauss',
'centigram',
'centihenry',
'centihertz',
'centihour',
'centihr',
'centijansky',
'centijoule',
'centikayser',
'centilightyear',
'centiliter',
'centilumen',
'centilux',
'centimeter',
'centiminute',
'centimole',
'centinewton',
'centiparsec',
'centipascal',
'centiphoton',
'centipixel',
'centipoise',
'centiradian',
'centirayleigh',
'centirydberg',
'centisecond',
'centisiemens',
'centisteradian',
'centistokes',
'centitesla',
'centivolt',
'centivoxel',
'centiwatt',
'centiweber',
'centiyear',
'cerg',
'cg',
'cgs',
'ch',
'chan',
'ck',
'cl',
'clm',
'clx',
'clyr',
'cm',
'cmag',
'cmin',
'cmol',
'cohm',
'core',
'coulomb',
'count',
'cpc',
'cph',
'cphoton',
'cpix',
'cpixel',
'crad',
'cs',
```

```
 'csr',
 'ct',
 'cu',
 'curie',
 'cvox',
 'cvoxel',
 'cy',
 'cycle',
 'cyr',
 'd',
 'dA',
 'dAU',
 'dB',
 'dBa',
 'dC',
 'dD',
 'dF',
 'dG',
 'dGal',
 'dH',
 'dHz',
 'dJ',
 'dJy',
 'dK',
 'dL',
 'dN',
 'dOhm',
 'dP',
 'dPa',
 'dR',
 'dRy',
 'dS',
 'dSt',
 ...]
```

To create a quantity, we multiply a value by a unit.

```
angle = 10 * u.degree
type(angle)
```

```
astropy.units.quantity.Quantity
```

The result is a `Quantity` object. Jupyter knows how to display `Quantities` like this:

```
angle
```

$10 \, ^{\circ}$

Quantities provide a method called `to` that converts to other units. For example, we can compute the number of arcminutes in `angle`:

```
angle_arcmin = angle.to(u.arcmin)
angle_arcmin
```

$600 \, ^{\prime}$

If you add quantities, Astropy converts them to compatible units, if possible:

```
angle + 30 * u.arcmin
```

$10.5 \, ^{\circ}$

If the units are not compatible, you get an error. For example:

```
angle + 5 * u.second
```

causes a `UnitConversionError`.

Exercise

Create a quantity that represents 5 arcminutes and assign it to a variable called `radius`.

Then convert it to degrees.

Selecting a Region

One of the most common ways to restrict a query is to select stars in a particular region of the sky. For example, here's a query from the Gaia archive documentation that selects objects in a circular region centered at (88.8, 7.4) with a search radius of 5 arcmin (0.08333 deg).

```
query_cone = """SELECT
TOP 10
source_id
FROM gaiadr2.gaia_source
WHERE 1=CONTAINS(
  POINT(ra, dec),
  CIRCLE(88.8, 7.4, 0.08333333))
"""
```

This query uses three keywords that are specific to ADQL (not SQL):

- `POINT`: a location in ICRS coordinates, specified in degrees of right ascension and declination.
- `CIRCLE`: a circle where the first two values are the coordinates of the center and the third is the radius in degrees.
- `CONTAINS`: a function that returns `1` if a `POINT` is contained in a shape and `0` otherwise.

Here is the documentation of `CONTAINS`.

A query like this is called a cone search because it selects stars in a cone. Here's how we run it.

```
from astroquery.gaia import Gaia

job = Gaia.launch_job(query_cone)
job
```

```
Created TAP+ (v1.2.1) - Connection:
        Host: gea.esac.esa.int
        Use HTTPS: True
        Port: 443
        SSL Port: 443
Created TAP+ (v1.2.1) - Connection:
        Host: geadata.esac.esa.int
        Use HTTPS: True
        Port: 443
        SSL Port: 443
```

```
<astroquery.utils.tap.model.job.Job at 0x7f277785fa30>
```

```
results = job.get_results()
results
```

Table length=10

source_id
int64
3322773965056065536
3322773758899157120
3322774068134271104
3322773930696320512
3322774377374425728
3322773724537891456
3322773724537891328
3322773930696321792
3322773724537890944
3322773930696322176

Exercise

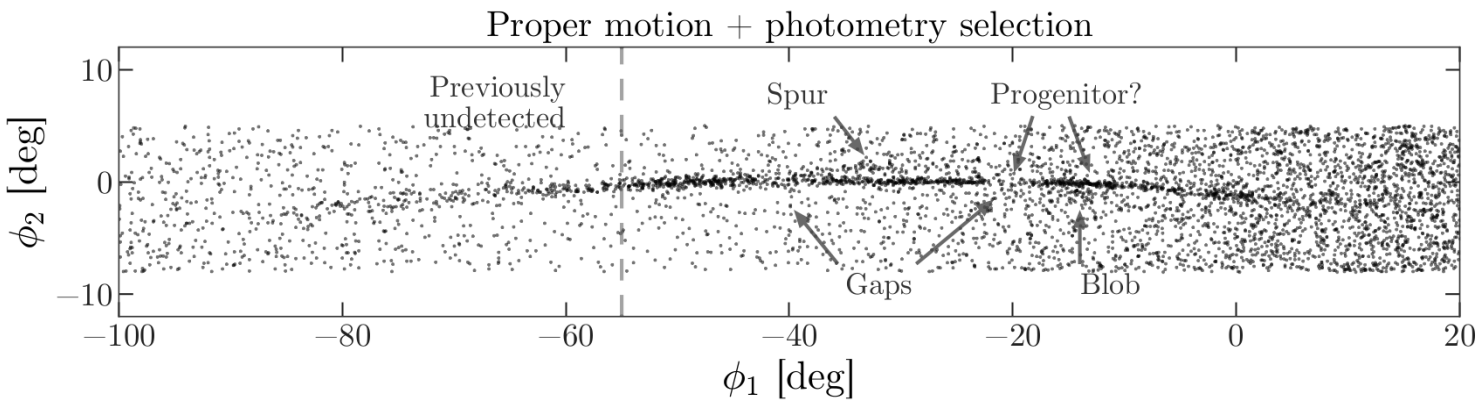
When you are debugging queries like this, you can use `TOP` to limit the size of the results, but then you still don't know how big the results will be.

An alternative is to use `COUNT`, which asks for the number of rows that would be selected, but it does not return them.

In the previous query, replace `TOP 10 source_id` with `COUNT(source_id)` and run the query again. How many stars has Gaia identified in the cone we searched?

Getting GD-1 Data

From the Price-Whelan and Bonaca paper, we will try to reproduce Figure 1, which includes this representation of stars likely to belong to GD-1:



The axes of this figure are defined so the x-axis is aligned with the stars in GD-1, and the y-axis is perpendicular.

- Along the x-axis (ϕ_1) the figure extends from -100 to 20 degrees.
- Along the y-axis (ϕ_2) the figure extends from about -8 to 4 degrees.

Ideally, we would select all stars from this rectangle, but there are more than 10 million of them, so

- That would be difficult to work with,
- As anonymous Gaia users, we are limited to 3 million rows in a single query, and
- While we are developing and testing code, it will be faster to work with a smaller dataset.

So we'll start by selecting stars in a smaller rectangle near the center of GD-1, from -55 to -45 degrees ϕ_1 and -8 to 4 degrees ϕ_2.

But first we let's see how to represent these coordinates with Astropy.

Transforming coordinates

Astropy provides a `SkyCoord` object that represents sky coordinates relative to a specified frame.

The following example creates a `SkyCoord` object that represents the approximate coordinates of Betelgeuse (alf Ori) in the ICRS frame.

ICRS is the "International Celestial Reference System", adopted in 1997 by the International Astronomical Union.

```
from astropy.coordinates import SkyCoord

ra = 88.8 * u.degree
dec = 7.4 * u.degree
coord_icrs = SkyCoord(ra=ra, dec=dec, frame='icrs')

coord_icrs
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    (88.8, 7.4)>
```

`SkyCoord` provides a function that transforms to other frames. For example, we can transform `coords_icrs` to Galactic coordinates like this:

```
coord_galactic = coord_icrs.transform_to('galactic')
coord_galactic
```

```
<SkyCoord (Galactic): (l, b) in deg
    (199.79693102, -8.95591653)>
```

Notice that in the Galactic frame, the coordinates are called `l` and `b`, not `ra` and `dec`.

To transform to and from GD-1 coordinates, we'll use a frame defined by Gala, which is an Astropy-affiliated library that provides tools for galactic dynamics.

Gala provides `GD1Koposov10`, which is "a Heliocentric spherical coordinate system defined by the orbit of the GD-1 stream".

```
from gala.coordinates import GD1Koposov10

gd1_frame = GD1Koposov10()
gd1_frame
```

```
<GD1Koposov10 Frame>
```

We can use it to find the coordinates of Betelgeuse in the GD-1 frame, like this:

```
coord_gd1 = coord_icrs.transform_to(gd1_frame)
coord_gd1
```

```
<SkyCoord (GD1Koposov10): (phi1, phi2) in deg
    (-94.97222038, 34.5813813)>
```

Notice that the coordinates are called `phi1` and `phi2`. These are the coordinates shown in the figure from the paper, above.

Exercise

Let's find the location of GD-1 in ICRS coordinates.

1. Create a `SkyCoord` object at 0°, 0° in the GD-1 frame.
2. Transform it to the ICRS frame.

Hint: Because ICRS is built into Astropy, you can specify it by name, `icrs` (as we did with `galactic`).

Notice that the origin of the GD-1 frame maps to `ra=200`, exactly, in ICRS. That's by design.

Selecting a rectangle

Now we'll use these coordinate transformations to define a rectangle in the GD-1 frame and transform it to ICRS.

The following variables define the boundaries of the rectangle in ϕ_1 and ϕ_2.

```
phi1_min = -55 * u.degree
phi1_max = -45 * u.degree
phi2_min = -8 * u.degree
phi2_max = 4 * u.degree
```

To create a rectangle, we'll use the following function, which takes the lower and upper bounds as parameters.

```
def make_rectangle(x1, x2, y1, y2):
    """Return the corners of a rectangle."""
    xs = [x1, x1, x2, x2, x1]
    ys = [y1, y2, y2, y1, y1]
    return xs, ys
```

The return value is a tuple containing a list of coordinates in `phi1` followed by a list of coordinates in `phi2`.

```
phi1_rect, phi2_rect = make_rectangle(
    phi1_min, phi1_max, phi2_min, phi2_max)
```

`phi1_rect` and `phi2_rect` contains the coordinates of the corners of a rectangle in the GD-1 frame.

In order to use them in a Gaia query, we have to convert them to ICRS. First we'll put them into a `SkyCoord` object.

```
corners = SkyCoord(phi1=phi1_rect, phi2=phi2_rect, frame=gd1_frame)
corners
```

```
<SkyCoord (GD1Koposov10): (phi1, phi2) in deg
    [(-55., -8.), (-55.,  4.), (-45.,  4.), (-45., -8.), (-55., -8.)]>
```

Now we can use `transform_to` to convert to ICRS coordinates.

```
corners_icrs = corners.transform_to('icrs')
corners_icrs
```

```
<SkyCoord (ICRS): (ra, dec) in deg
    [(146.27533314, 19.26190982), (135.42163944, 25.87738723),
     (141.60264825, 34.3048303 ), (152.81671045, 27.13611254),
     (146.27533314, 19.26190982)]>
```

Notice that a rectangle in one coordinate system is not necessarily a rectangle in another. In this example, the result is a (non-rectangular) polygon.

Defining a polygon

In order to use this polygon as part of an ADQL query, we have to convert it to a string with a comma-separated list of coordinates, as in this example:

```
"""
POLYGON(143.65, 20.98,
        134.46, 26.39,
        140.58, 34.85,
        150.16, 29.01)
"""
```

`SkyCoord` provides `to_string`, which produces a list of strings.

```
t = corners_icrs.to_string()
t
```

```
['146.275 19.2619',
 '135.422 25.8774',
 '141.603 34.3048',
 '152.817 27.1361',
 '146.275 19.2619']
```

We can use the Python string function `join` to join `t` into a single string (with spaces between the pairs):

```
s = ' '.join(t)
s
```

```
'146.275 19.2619 135.422 25.8774 141.603 34.3048 152.817 27.1361 146.275 19.2619'
```

That's almost what we need, but we have to replace the spaces with commas.

```
s.replace(' ', ', ')
```

```
'146.275, 19.2619, 135.422, 25.8774, 141.603, 34.3048, 152.817, 27.1361, 146.275, 19.2619'
```

The following function combines these steps.

```
def skycoord_to_string(skycoord):
    """Convert SkyCoord to string."""
    t = skycoord.to_string()
    s = ' '.join(t)
    return s.replace(' ', ', ')
```

Here's how we use it.

```
point_list = skycoord_to_string(corners_icrs)
point_list
```

```
'146.275, 19.2619, 135.422, 25.8774, 141.603, 34.3048, 152.817, 27.1361, 146.275, 19.2619'
```

Assembling the query

Now we're ready to assemble the query. We need `columns` again (as we saw in the previous lesson).

```
columns = 'source_id, ra, dec, pmra, pmdec, parallax'
```

And here's the query base we used in the previous lesson:

```
query3_base = """SELECT
TOP 10
{columns}
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
"""
```

Now we'll add a `WHERE` clause to select stars in the polygon we defined.

```
query4_base = """SELECT
TOP 10
{columns}
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
  AND 1 = CONTAINS(POINT(ra, dec),
                   POLYGON({point_list}))
"""
```

The query base contains format specifiers for `columns` and `point_list`.

We'll use `format` to fill in these values.

```
query4 = query4_base.format(columns=columns,
                            point_list=point_list)
print(query4)
```

```
SELECT
TOP 10
source_id, ra, dec, pmra, pmdec
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
  AND 1 = CONTAINS(POINT(ra, dec),
                   POLYGON(146.275, 19.2619, 135.422, 25.8774, 141.603, 34.3048, 152.817,
27.1361, 146.275, 19.2619))
```

As always, we should take a minute to proof-read the query before we launch it.

```
job = Gaia.launch_job_async(query4)
print(job)
```

```
INFO: Query finished. [astroquery.utils.tap.core]
<Table length=10>
   name    dtype    unit                                 description
--------- ------- -------- -------------------------------------------------------------------
source_id   int64          Unique source identifier (unique within a particular Data Release)
       ra float64      deg                                                        Right ascension
      dec float64      deg                                                            Declination
     pmra float64 mas / yr                          Proper motion in right ascension direction
    pmdec float64 mas / yr                              Proper motion in declination direction
Jobid: 16158158738080
Phase: COMPLETED
Owner: None
Output file: async_20210315094433.vot
Results: None
```

Here are the results.

```
results = job.get_results()
results
```

Table length=10

source_id	ra	dec	pmra
	deg	deg	mas / yr
int64	float64	float64	float64
637987125186749568	142.48301935991023	21.75771616932985	-2.5168384683875766
638285195917112960	142.25452941346344	22.476168171141378	2.6627020143457996
638073505568978688	142.64528557468074	22.16693224953078	18.30674739454163
638086386175786752	142.57739430926034	22.22791951401365	0.9877856720147953
638049655615392384	142.58913564478618	22.110783166677418	0.24443878227817095
638267565075964032	141.81762228999614	22.375696125322275	-3.413174589660796
638028902333511168	143.18339801317677	22.2512465812369	7.848511762712128
638085767700610432	142.9347319464589	22.46244080823965	-3.6585960944321476
638299863230178304	142.26769745823267	22.640183776884836	-1.8168370892218297
637973067758974208	142.89551292869012	21.612824100339875	-8.645166256559063

Finally, we can remove `TOP 10` run the query again.

The result is bigger than our previous queries, so it will take a little longer.

```
query5_base = """SELECT
{columns}
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
  AND 1 = CONTAINS(POINT(ra, dec),
                   POLYGON({point_list}))
"""
```

```
query5 = query5_base.format(columns=columns,
                            point_list=point_list)
print(query5)
```

```
SELECT
source_id, ra, dec, pmra, pmdec
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
  AND 1 = CONTAINS(POINT(ra, dec),
                   POLYGON(146.275, 19.2619, 135.422, 25.8774, 141.603, 34.3048, 152.817,
27.1361, 146.275, 19.2619))
```

```
job = Gaia.launch_job_async(query5)
print(job)
```

```
INFO: Query finished. [astroquery.utils.tap.core]
<Table length=140339>
   name    dtype    unit                              description
--------- ------- -------- ------------------------------------------------------------------
source_id   int64          Unique source identifier (unique within a particular Data Release)
       ra float64      deg                                                      Right ascension
      dec float64      deg                                                          Declination
     pmra float64 mas / yr                          Proper motion in right ascension direction
    pmdec float64 mas / yr                              Proper motion in declination direction
Jobid: 16158158867070
Phase: COMPLETED
Owner: None
Output file: async_20210315094446.vot
Results: None
```

```
results = job.get_results()
len(results)
```

```
140339
```

There are more than 100,000 stars in this polygon, but that's a manageable size to work with.

Saving results

This is the set of stars we'll work with in the next step. But since we have a substantial dataset now, this is a good time to save it.

Storing the data in a file means we can shut down this notebook and pick up where we left off without running the previous query again.

Astropy `Table` objects provide `write`, which writes the table to disk.

```
filename = 'gd1_results.fits'
results.write(filename, overwrite=True)
```

Because the filename ends with `fits`, the table is written in the FITS format, which preserves the metadata associated with the table.

If the file already exists, the `overwrite` argument causes it to be overwritten.

We can use `getsize` to confirm that the file exists and check the size:

```
from os.path import getsize

MB = 1024 * 1024
getsize(filename) / MB
```

```
5.36407470703125
```

Summary

In this notebook, we composed more complex queries to select stars within a polygonal region of the sky. Then we downloaded the results and saved them in a FITS file.

In the next notebook, we'll reload the data from this file and replicate the next step in the analysis, using proper motion to identify stars likely to be in GD-1.

Best practices

- For measurements with units, use `Quantity` objects that represent units explicitly and check for errors.
- Use the `format` function to compose queries; code written this way is easier to read and less error-prone.
- Develop queries incrementally: start with something simple, test it, and add a little bit at a time.
- Once you have a query working, save the data in a local file. If you shut down the notebook and come back to it later, you can reload the file; you don't have to run the query again.
