Astronomical Data in Python

Allen B. Downey

Nov 04, 2020

CONTENTS

Astronomical Data in Python is an introduction to tools and practices for working with astronomical data. Topics covered include:

- Writing queries that select and download data from a database.
- Using data stored in an Astropy `Table` or Pandas `DataFrame`.
- Working with coordinates and other quantities with units.
- Storing data in various formats.
- Performing database join operations that combine data from multiple tables.
- Visualizing data and preparing publication-quality figures.

As a running example, we will replicate part of the analysis in a recent paper, "Off the beaten path: Gaia reveals GD-1 stars outside of the main stream" by Adrian M. Price-Whelan and Ana Bonaca.

This material was developed in collaboration with The Carpentries and the Astronomy Curriculum Development Committee, and supported by funding from the American Institute of Physics through the American Astronomical Society.

I am grateful for contributions from the members of the committee – Azalee Bostroem, Rodolfo Montez, and Phil Rosenfield – and from Erin Becker, Brett Morris and Adrian Price-Whelan.

The original format of this material is a series of Jupyter notebooks. Using the links below, you can read the notebooks on NBViewer or run them on Colab. If you want to run the notebooks in your own environment, you can download them from this repository and follow the instructions below to set up your environment.

This material is also available in the form of Carpentries lessons, but you should be aware that these versions might diverge in the future.

Prerequisites

This material should be accessible to people familiar with basic Python, but not necessarily the libraries we will use, like Astropy or Pandas. If you are familiar with Python lists and dictionaries, and you know how to write a function that takes parameters and returns a value, that should be enough.

We assume that you are familiar with astronomy at the undergraduate level, but we will not assume specialized knowledge of the datasets or analysis methods we'll use.

Notebook 1

This notebook demonstrates the following steps:

1. Making a connection to the Gaia server,
2. Exploring information about the database and the tables it contains,
3. Writing a query and sending it to the server, and finally
4. Downloading the response from the server as an Astropy `Table`.

Press this button to run this notebook on Colab:

or click here to read it on NBViewer

Notebook 2

This notebook starts 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:

- Use `Quantity` objects to represent measurements with units.

- Use the `Gala` library 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.

Press this button to run this notebook on Colab:

or click here to read it on NBViewer

Notebook 3

Here are the steps in this notebook:

1. We'll read back the results from the previous notebook, which we saved in a FITS file.
2. Then we'll transform the coordinates and proper motion data from ICRS back to the coordinate frame of GD-1.
3. We'll put those results into a Pandas `DataFrame`, which we'll use to select stars near the centerline of GD-1.
4. Plotting the proper motion of those stars, we'll identify a region of proper motion for stars that are likely to be in GD-1.
5. Finally, we'll select and plot the stars whose proper motion is in that region.

Press this button to run this notebook on Colab:

or click here to read it on NBViewer

Notebook 4

Here are the steps in this notebook:

1. Using data from the previous notebook, we'll identify the values of proper motion for stars likely to be in GD-1.
2. Then we'll compose an ADQL query that selects stars based on proper motion, so we can download only the data we need.
3. We'll also see how to write the results to a CSV file.

That will make it possible to search a bigger region of the sky in a single query.

Press this button to run this notebook on Colab:

or click here to read it on NBViewer

Notebook 5

Here are the steps in this notebook:

1. We'll reload the candidate stars we identified in the previous notebook.
2. Then we'll run a query on the Gaia server that uploads the table of candidates and uses a `JOIN` operation to select photometry data for the candidate stars.
3. We'll write the results to a file for use in the next notebook.

Press this button to run this notebook on Colab:

or click here to read it on NBViewer

Notebook 6

Here are the steps in this notebook:

1. We'll reload the data from the previous notebook and make a color-magnitude diagram.
2. Then we'll specify a polygon in the diagram that contains stars with the photometry we expect.
3. Then we'll merge the photometry data with the list of candidate stars, storing the result in a Pandas `DataFrame`.

Press this button to run this notebook on Colab:

or click here to read it on NBViewer

Notebook 7

Here are the steps in this notebook:

1. Starting with the figure from the previous notebook, we'll add annotations to present the results more clearly.
2. The we'll see several ways to customize figures to make them more appealing and effective.
3. Finally, we'll see how to make a figure with multiple panels or subplots.

Press this button to run this notebook on Colab:

or click here to read it on NBViewer

Installation instructions

Coming soon.

CHAPTER

ONE

CHAPTER 1

Astronomical Data in Python is an introduction to tools and practices for working with astronomical data. Topics covered include:

- Writing queries that select and download data from a database.
- Using data stored in an Astropy `Table` or Pandas `DataFrame`.
- Working with coordinates and other quantities with units.
- Storing data in various formats.
- Performing database join operations that combine data from multiple tables.
- Visualizing data and preparing publication-quality figures.

As a running example, we will replicate part of the analysis in a recent paper, "Off the beaten path: Gaia reveals GD-1 stars outside of the main stream" by Adrian M. Price-Whelan and Ana Bonaca.

As the abstract explains, "Using data from the Gaia second data release combined with Pan-STARRS photometry, we present a sample of highly-probable members of the longest cold stream in the Milky Way, GD-1."

GD-1 is a stellar stream, which is "an association of stars orbiting a galaxy that was once a globular cluster or dwarf galaxy that has now been torn apart and stretched out along its orbit by tidal forces."

This article in *Science* magazine explains some of the background, including the process that led to the paper and an discussion of the scientific implications:

- "The streams are particularly useful for … galactic archaeology — rewinding the cosmic clock to reconstruct the assembly of the Milky Way."
- "They also are being used as exquisitely sensitive scales to measure the galaxy's mass."
- "… the streams are well-positioned to reveal the presence of dark matter … because the streams are so fragile, theorists say, collisions with marauding clumps of dark matter could leave telltale scars, potential clues to its nature."

1.1 Data

The datasets we will work with are:

- Gaia, which is "a space observatory of the European Space Agency (ESA), launched in 2013 … designed for astrometry: measuring the positions, distances and motions of stars with unprecedented precision", and
- Pan-STARRS, The Panoramic Survey Telescope and Rapid Response System, which is a survey designed to monitor the sky for transient objects, producing a catalog with accurate astronometry and photometry of detected sources.

Both of these datasets are very large, which can make them challenging to work with. It might not be possible, or practical, to download the entire dataset. One of the goals of this workshop is to provide tools for working with large datasets.

1.2 Prerequisites

These notebooks are meant for people who are familiar with basic Python, but not necessarily the libraries we will use, like Astropy or Pandas. If you are familiar with Python lists and dictionaries, and you know how to write a function that takes parameters and returns a value, you know enough Python to get started.

We assume that you have some familiarity with operating systems, like the ability to use a command-line interface. But we don't assume you have any prior experience with databases.

We assume that you are familiar with astronomy at the undergraduate level, but we will not assume specialized knowledge of the datasets or analysis methods we'll use.

1.3 Outline

The first lesson demonstrates the steps for selecting and downloading data from the Gaia Database:

1. First we'll make a connection to the Gaia server,
2. We will explore information about the database and the tables it contains,
3. We will write a query and send it to the server, and finally
4. We will download the response from the server.

After completing this lesson, you should be able to

- Compose a basic query in ADQL.
- Use queries to explore a database and its tables.
- Use queries to download data.
- Develop, test, and debug a query incrementally.

1.4 Query Language

In order to select data from a database, you have to compose a query, which is like a program written in a "query language". The query language we'll use is ADQL, which stands for "Astronomical Data Query Language".

ADQL is a dialect of SQL (Structured Query Language), which is by far the most commonly used query language. Almost everything you will learn about ADQL also works in SQL.

The reference manual for ADQL is here. But you might find it easier to learn from this ADQL Cookbook.

1.5 Installing libraries

The library we'll use to get Gaia data is Astroquery.

If you are running this notebook on Colab, you can run the following cell to install Astroquery and the other libraries we'll use.

If you are running this notebook on your own computer, you might have to install these libraries yourself.

If you are using this notebook as part of a Carpentries workshop, you should have received setup instructions.

TODO: Add a link to the instructions.

```
# If we're running on Colab, install libraries

import sys
IN_COLAB = 'google.colab' in sys.modules

if IN_COLAB:
    !pip install astroquery astro-gala pyia
```

1.6 Connecting to Gaia

Astroquery provides `Gaia`, which is an object that represents a connection to the Gaia database.

We can connect to the Gaia database like this:

```
from astroquery.gaia import Gaia
```

```
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
```

Running this import statement has the effect of creating a TAP+ connection; TAP stands for "Table Access Protocol". It is a network protocol for sending queries to the database and getting back the results. We're not sure why it seems to create two connections.

1.7 Databases and Tables

What is a database, anyway? Most generally, it can be any collection of data, but when we are talking about ADQL or SQL:

- A database is a collection of one or more named tables.
- Each table is a 2-D array with one or more named columns of data.

We can use `Gaia.load_tables` to get the names of the tables in the Gaia database. With the option `only_names=True`, it loads information about the tables, called the "metadata", not the data itself.

```
tables = Gaia.load_tables(only_names=True)
```

```
INFO: Retrieving tables... [astroquery.utils.tap.core]
INFO: Parsing tables... [astroquery.utils.tap.core]
INFO: Done. [astroquery.utils.tap.core]
```

```
for table in (tables):
    print(table.get_qualified_name())
```

```
external.external.apassdr9
external.external.gaiadr2_geometric_distance
external.external.galex_ais
external.external.ravedr5_com
external.external.ravedr5_dr5
external.external.ravedr5_gra
external.external.ravedr5_on
external.external.sdssdr13_photoprimary
external.external.skymapperdr1_master
external.external.tmass_xsc
public.public.hipparcos
public.public.hipparcos_newreduction
public.public.hubble_sc
public.public.igsl_source
public.public.igsl_source_catalog_ids
public.public.tycho2
public.public.dual
tap_config.tap_config.coord_sys
tap_config.tap_config.properties
tap_schema.tap_schema.columns
tap_schema.tap_schema.key_columns
tap_schema.tap_schema.keys
tap_schema.tap_schema.schemas
tap_schema.tap_schema.tables
gaiadr1.gaiadr1.aux_qso_icrf2_match
gaiadr1.gaiadr1.ext_phot_zero_point
gaiadr1.gaiadr1.allwise_best_neighbour
gaiadr1.gaiadr1.allwise_neighbourhood
gaiadr1.gaiadr1.gsc23_best_neighbour
gaiadr1.gaiadr1.gsc23_neighbourhood
gaiadr1.gaiadr1.ppmxl_best_neighbour
gaiadr1.gaiadr1.ppmxl_neighbourhood
gaiadr1.gaiadr1.sdss_dr9_best_neighbour
gaiadr1.gaiadr1.sdss_dr9_neighbourhood
gaiadr1.gaiadr1.tmass_best_neighbour
gaiadr1.gaiadr1.tmass_neighbourhood
gaiadr1.gaiadr1.ucac4_best_neighbour
gaiadr1.gaiadr1.ucac4_neighbourhood
gaiadr1.gaiadr1.urat1_best_neighbour
gaiadr1.gaiadr1.urat1_neighbourhood
gaiadr1.gaiadr1.cepheid
gaiadr1.gaiadr1.phot_variable_time_series_gfov
gaiadr1.gaiadr1.phot_variable_time_series_gfov_statistical_parameters
gaiadr1.gaiadr1.rrlyrae
gaiadr1.gaiadr1.variable_summary
gaiadr1.gaiadr1.allwise_original_valid
gaiadr1.gaiadr1.gsc23_original_valid
```

(continues on next page)

(continued from previous page)

```
gaiadr1.gaiadr1.ppmxl_original_valid
gaiadr1.gaiadr1.sdssdr9_original_valid
gaiadr1.gaiadr1.tmass_original_valid
gaiadr1.gaiadr1.ucac4_original_valid
gaiadr1.gaiadr1.urat1_original_valid
gaiadr1.gaiadr1.gaia_source
gaiadr1.gaiadr1.tgas_source
gaiadr2.gaiadr2.aux_allwise_agn_gdr2_cross_id
gaiadr2.gaiadr2.aux_iers_gdr2_cross_id
gaiadr2.gaiadr2.aux_sso_orbit_residuals
gaiadr2.gaiadr2.aux_sso_orbits
gaiadr2.gaiadr2.dr1_neighbourhood
gaiadr2.gaiadr2.allwise_best_neighbour
gaiadr2.gaiadr2.allwise_neighbourhood
gaiadr2.gaiadr2.apassdr9_best_neighbour
gaiadr2.gaiadr2.apassdr9_neighbourhood
gaiadr2.gaiadr2.gsc23_best_neighbour
gaiadr2.gaiadr2.gsc23_neighbourhood
gaiadr2.gaiadr2.hipparcos2_best_neighbour
gaiadr2.gaiadr2.hipparcos2_neighbourhood
gaiadr2.gaiadr2.panstarrs1_best_neighbour
gaiadr2.gaiadr2.panstarrs1_neighbourhood
gaiadr2.gaiadr2.ppmxl_best_neighbour
gaiadr2.gaiadr2.ppmxl_neighbourhood
gaiadr2.gaiadr2.ravedr5_best_neighbour
gaiadr2.gaiadr2.ravedr5_neighbourhood
gaiadr2.gaiadr2.sdssdr9_best_neighbour
gaiadr2.gaiadr2.sdssdr9_neighbourhood
gaiadr2.gaiadr2.tmass_best_neighbour
gaiadr2.gaiadr2.tmass_neighbourhood
gaiadr2.gaiadr2.tycho2_best_neighbour
gaiadr2.gaiadr2.tycho2_neighbourhood
gaiadr2.gaiadr2.urat1_best_neighbour
gaiadr2.gaiadr2.urat1_neighbourhood
gaiadr2.gaiadr2.sso_observation
gaiadr2.gaiadr2.sso_source
gaiadr2.gaiadr2.vari_cepheid
gaiadr2.gaiadr2.vari_classifier_class_definition
gaiadr2.gaiadr2.vari_classifier_definition
gaiadr2.gaiadr2.vari_classifier_result
gaiadr2.gaiadr2.vari_long_period_variable
gaiadr2.gaiadr2.vari_rotation_modulation
gaiadr2.gaiadr2.vari_rrlyrae
gaiadr2.gaiadr2.vari_short_timescale
gaiadr2.gaiadr2.vari_time_series_statistics
gaiadr2.gaiadr2.panstarrs1_original_valid
gaiadr2.gaiadr2.gaia_source
gaiadr2.gaiadr2.ruwe
```

So that's a lot of tables. The ones we'll use are:

- `gaiadr2.gaia_source`, which contains Gaia data from data release 2,
- `gaiadr2.panstarrs1_original_valid`, which contains the photometry data we'll use from PanSTARRS, and
- `gaiadr2.panstarrs1_best_neighbour`, which we'll use to cross-match each star observed by Gaia with the same star observed by PanSTARRS.

We can use `load_table` (not `load_tables`) to get the metadata for a single table. The name of this function is misleading, because it only downloads metadata.

```
meta = Gaia.load_table('gaiadr2.gaia_source')
meta
```

```
Retrieving table 'gaiadr2.gaia_source'
Parsing table 'gaiadr2.gaia_source'...
Done.
```

```
<astroquery.utils.tap.model.taptable.TapTableMeta at 0x7f922376e0a0>
```

Jupyter shows that the result is an object of type `TapTableMeta`, but it does not display the contents.

To see the metadata, we have to print the object.

```
print(meta)
```

```
TAP Table name: gaiadr2.gaiadr2.gaia_source
Description: This table has an entry for every Gaia observed source as listed in the
Main Database accumulating catalogue version from which the catalogue
release has been generated. It contains the basic source parameters,
that is only final data (no epoch data) and no spectra (neither final
nor epoch).
Num. columns: 96
```

Notice one gotcha: in the list of table names, this table appears as `gaiadr2.gaiadr2.gaia_source`, but when we load the metadata, we refer to it as `gaiadr2.gaia_source`.

Exercise: Go back and try

```
meta = Gaia.load_table('gaiadr2.gaiadr2.gaia_source')
```

What happens? Is the error message helpful? If you had not made this error deliberately, would you have been able to figure it out?

1.8 Columns

The following loop prints the names of the columns in the table.

```
for column in meta.columns:
    print(column.name)
```

```
solution_id
designation
source_id
random_index
ref_epoch
ra
ra_error
dec
dec_error
parallax
parallax_error
```

(continues on next page)

(continued from previous page)

```
parallax_over_error
pmra
pmra_error
pmdec
pmdec_error
ra_dec_corr
ra_parallax_corr
ra_pmra_corr
ra_pmdec_corr
dec_parallax_corr
dec_pmra_corr
dec_pmdec_corr
parallax_pmra_corr
parallax_pmdec_corr
pmra_pmdec_corr
astrometric_n_obs_al
astrometric_n_obs_ac
astrometric_n_good_obs_al
astrometric_n_bad_obs_al
astrometric_gof_al
astrometric_chi2_al
astrometric_excess_noise
astrometric_excess_noise_sig
astrometric_params_solved
astrometric_primary_flag
astrometric_weight_al
astrometric_pseudo_colour
astrometric_pseudo_colour_error
mean_varpi_factor_al
astrometric_matched_observations
visibility_periods_used
astrometric_sigma5d_max
frame_rotator_object_type
matched_observations
duplicated_source
phot_g_n_obs
phot_g_mean_flux
phot_g_mean_flux_error
phot_g_mean_flux_over_error
phot_g_mean_mag
phot_bp_n_obs
phot_bp_mean_flux
phot_bp_mean_flux_error
phot_bp_mean_flux_over_error
phot_bp_mean_mag
phot_rp_n_obs
phot_rp_mean_flux
phot_rp_mean_flux_error
phot_rp_mean_flux_over_error
phot_rp_mean_mag
phot_bp_rp_excess_factor
phot_proc_mode
bp_rp
bp_g
g_rp
radial_velocity
radial_velocity_error
```

(continues on next page)

(continued from previous page)

```
rv_nb_transits
rv_template_teff
rv_template_logg
rv_template_fe_h
phot_variable_flag
l
b
ecl_lon
ecl_lat
priam_flags
teff_val
teff_percentile_lower
teff_percentile_upper
a_g_val
a_g_percentile_lower
a_g_percentile_upper
e_bp_min_rp_val
e_bp_min_rp_percentile_lower
e_bp_min_rp_percentile_upper
flame_flags
radius_val
radius_percentile_lower
radius_percentile_upper
lum_val
lum_percentile_lower
lum_percentile_upper
datalink_url
epoch_photometry_url
```

You can probably guess what many of these columns are by looking at the names, but you should resist the temptation to guess. To find out what the columns mean, read the documentation.

If you want to know what can go wrong when you don't read the documentation, you might like this article.

Exercise: One of the other tables we'll use is `gaiadr2.gaiadr2.panstarrs1_original_valid`. Use `load_table` to get the metadata for this table. How many columns are there and what are their names?

Hint: Remember the gotcha we mentioned earlier.

```
# Solution

for column in meta2.columns:
    print(column.name)
```

```
obj_name
obj_id
ra
dec
ra_error
dec_error
epoch_mean
g_mean_psf_mag
g_mean_psf_mag_error
g_flags
r_mean_psf_mag
r_mean_psf_mag_error
r_flags
```

(continues on next page)

(continued from previous page)

```
i_mean_psf_mag
i_mean_psf_mag_error
i_flags
z_mean_psf_mag
z_mean_psf_mag_error
z_flags
y_mean_psf_mag
y_mean_psf_mag_error
y_flags
n_detections
zone_id
obj_info_flag
quality_flag
```

1.9 Writing queries

By now you might be wondering how we actually download the data. With tables this big, you generally don't. Instead, you use queries to select only the data you want.

A query is a string written in a query language like SQL; for the Gaia database, the query language is a dialect of SQL called ADQL.

Here's an example of an ADQL query.

```
query1 = """SELECT
TOP 10
source_id, ref_epoch, ra, dec, parallax
FROM gaiadr2.gaia_source"""
```

Python note: We use a triple-quoted string here so we can include line breaks in the query, which makes it easier to read.

The words in uppercase are ADQL keywords:

- `SELECT` indicates that we are selecting data (as opposed to adding or modifying data).
- `TOP` indicates that we only want the first 10 rows of the table, which is useful for testing a query before asking for all of the data.
- `FROM` specifies which table we want data from.

The third line is a list of column names, indicating which columns we want.

In this example, the keywords are capitalized and the column names are lowercase. This is a common style, but it is not required. ADQL and SQL are not case-sensitive.

To run this query, we use the `Gaia` object, which represents our connection to the Gaia database, and invoke `launch_job`:

```
job1 = Gaia.launch_job(query1)
job1
```

```
<astroquery.utils.tap.model.job.Job at 0x7f9222e9cb20>
```

The result is an object that represents the job running on a Gaia server.

If you print it, it displays metadata for the forthcoming table.

```
print(job1)
```

```
<Table length=10>
   name    dtype  unit                              description
--------- ------- ---- ------------------------------------------------------------
↪---
source_id   int64      Unique source identifier (unique within a particular Data␣
↪Release)
ref_epoch float64   yr                                                  Reference␣
↪epoch
       ra float64  deg                                                  Right␣
↪ascension
      dec float64  deg                                                     ␣
↪Declination
 parallax float64  mas                                                       ␣
↪Parallax
Jobid: None
Phase: COMPLETED
Owner: None
Output file: sync_20201005090721.xml.gz
Results: None
```

Don't worry about `Results: None`. That does not actually mean there are no results.

However, `Phase: COMPLETED` indicates that the job is complete, so we can get the results like this:

```
results1 = job1.get_results()
type(results1)
```

```
astropy.table.table.Table
```

Optional detail: Why is `table` repeated three times? The first is the name of the module, the second is the name of the submodule, and the third is the name of the class. Most of the time we only care about the last one. It's like the Linnean name for gorilla, which is *Gorilla Gorilla Gorilla*.

The result is an Astropy Table, which is similar to a table in an SQL database except:

- SQL databases are stored on disk drives, so they are persistent; that is, they "survive" even if you turn off the computer. An Astropy `Table` is stored in memory; it disappears when you turn off the computer (or shut down this Jupyter notebook).
- SQL databases are designed to process queries. An Astropy `Table` can perform some query-like operations, like selecting columns and rows. But these operations use Python syntax, not SQL.

Jupyter knows how to display the contents of a `Table`.

```
results1
```

```
<Table length=10>
     source_id      ref_epoch ...        dec               parallax
                        yr     ...        deg                 mas
       int64         float64  ...      float64             float64
------------------- --------- ... ------------------ --------------------
4530738361793769600    2015.5 ...  20.40682117430378   0.9785380604519425
4530752651135081216    2015.5 ... 20.523350496351846   0.2674800612552977
4530743343951405568    2015.5 ... 20.474147574053124 -0.43911323550176806
4530755060627162368    2015.5 ... 20.558523922346158   1.1422630184554958
```

(continues on next page)

(continued from previous page)

```
4530746844341315968    2015.5 ... 20.377852388898184   1.0092247424630945
4530768456615026432    2015.5 ...  20.31829694530366 -0.06900136127674149
4530763513119137280    2015.5 ...  20.20956829578524   0.1266016679823622
4530736364618539264    2015.5 ... 20.346579041327693   0.3894019486060072
4530735952305177728    2015.5 ... 20.311030903719928   0.2041189982608354
4530751281056022656    2015.5 ... 20.460309556214753  0.10294642821734962
```

Each column has a name, units, and a data type.

For example, the units of `ra` and `dec` are degrees, and their data type is `float64`, which is a 64-bit floating-point number, used to store measurements with a fraction part.

This information comes from the Gaia database, and has been stored in the Astropy `Table` by Astroquery.

Exercise: Read the documentation of this table and choose a column that looks interesting to you. Add the column name to the query and run it again. What are the units of the column you selected? What is its data type?

1.10 Asynchronous queries

`launch_job` asks the server to run the job "synchronously", which normally means it runs immediately. But synchronous jobs are limited to 2000 rows. For queries that return more rows, you should run "asynchronously", which mean they might take longer to get started.

If you are not sure how many rows a query will return, you can use the SQL command `COUNT` to find out how many rows are in the result without actually returning them. We'll see an example of this later.

The results of an asynchronous query are stored in a file on the server, so you can start a query and come back later to get the results.

For anonymous users, files are kept for three days.

As an example, let's try a query that's similar to `query1`, with two changes:

- It selects the first 3000 rows, so it is bigger than we should run synchronously.
- It uses a new keyword, `WHERE`.

```
query2 = """SELECT TOP 3000
source_id, ref_epoch, ra, dec, parallax
FROM gaiadr2.gaia_source
WHERE parallax < 1
"""
```

A `WHERE` clause indicates which rows we want; in this case, the query selects only rows "where" `parallax` is less than 1. This has the effect of selecting stars with relatively low parallax, which are farther away. We'll use this clause to exclude nearby stars that are unlikely to be part of GD-1.

`WHERE` is one of the most common clauses in ADQL/SQL, and one of the most useful, because it allows us to select only the rows we need from the database.

We use `launch_job_async` to submit an asynchronous query.

```
job2 = Gaia.launch_job_async(query2)
print(job2)
```

```
INFO: Query finished. [astroquery.utils.tap.core]
<Table length=3000>
   name    dtype  unit                              description
--------- ------- ---- ------------------------------------------------------------
↪---
source_id   int64      Unique source identifier (unique within a particular Data␣
↪Release)
ref_epoch float64   yr                                                    Reference␣
↪epoch
       ra float64  deg                                                    Right␣
↪ascension
      dec float64  deg                                                       ␣
↪Declination
 parallax float64  mas                                                        ␣
↪Parallax
Jobid: 16019032422219O
Phase: COMPLETED
Owner: None
Output file: async_20201005090722.vot
Results: None
```

And here are the results.

```
results2 = job2.get_results()
results2
```

```
<Table length=3000>
     source_id      ref_epoch ...        dec               parallax
                        yr     ...        deg                 mas
       int64         float64  ...      float64             float64
------------------- --------- ... ------------------ --------------------
4530738361793769600    2015.5 ...  20.40682117430378   0.9785380604519425
4530752651135081216    2015.5 ... 20.523350496351846   0.2674800612552977
4530743343951405568    2015.5 ... 20.474147574053124 -0.43911323550176806
4530768456615026432    2015.5 ...  20.31829694530366 -0.06900136127674149
4530763513119137280    2015.5 ...  20.20956829578524   0.1266016679823622
4530736364618539264    2015.5 ... 20.346579041327693   0.3894019486060072
4530735952305177728    2015.5 ... 20.311030903719928   0.2041189982608354
4530751281056022656    2015.5 ... 20.460309556214753  0.10294642821734962
4530740938774409344    2015.5 ... 20.436140058941206   0.9242670062090182
                ...       ... ...                ...                  ...
4467710915011802624    2015.5 ... 1.1429085038160882  0.42361471245557913
4467706551328679552    2015.5 ... 1.0565747323689927    0.922888231734588
4467712255037300096    2015.5 ... 0.6581664892880896   -2.669179465293931
4467735001181761792    2015.5 ... 0.8947079323599124   0.6117399163086398
4467737101421916672    2015.5 ... 0.9806225910160181 -0.39818224846127004
4467707547757327488    2015.5 ... 1.0212759940136962   0.7741412301054209
4467732772094573056    2015.5 ... 0.9037072088489417  -1.7920417800164183
4467732355491087744    2015.5 ... 0.9197224705139885  -0.3464446494840354
4467717099766944512    2015.5 ...  0.726277659009568  0.05443955111134051
4467719058265781248    2015.5 ... 0.8205551921782785   0.3733943917490343
```

You might notice that some values of `parallax` are negative. As this FAQ explains, "Negative parallaxes are caused by errors in the observations." Negative parallaxes have "no physical meaning," but they can be a "useful diagnostic on the quality of the astrometric solution."

Later we will see an example where we use `parallax` and `parallax_error` to identify stars where the distance estimate is likely to be inaccurate.

Exercise: The clauses in a query have to be in the right order. Go back and change the order of the clauses in `query2` and run it again.

The query should fail, but notice that you don't get much useful debugging information.

For this reason, developing and debugging ADQL queries can be really hard. A few suggestions that might help:

- Whenever possible, start with a working query, either an example you find online or a query you have used in the past.
- Make small changes and test each change before you continue.
- While you are debugging, use `TOP` to limit the number of rows in the result. That will make each attempt run faster, which reduces your testing time.
- Launching test queries synchronously might make them start faster, too.

1.11 Operators

In a `WHERE` clause, you can use any of the SQL comparison operators; here are the most common ones:

Symbol	Operation
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
=	equal
!= or <>	not equal

Most of these are the same as Python, but some are not. In particular, notice that the equality operator is =, not ==. Be careful to keep your Python out of your ADQL!

You can combine comparisons using the logical operators:

- AND: true if both comparisons are true
- OR: true if either or both comparisons are true

Finally, you can use `NOT` to invert the result of a comparison.

Exercise: Read about SQL operators here and then modify the previous query to select rows where `bp_rp` is between `-0.75` and `2`.

You can read about this variable here.

```
# Solution

# This is what most people will probably do

query = """SELECT TOP 10
source_id, ref_epoch, ra, dec, parallax
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp > -0.75 AND bp_rp < 2
"""
```

```
# Solution

# But if someone notices the BETWEEN operator,
# they might do this

query = """SELECT TOP 10
source_id, ref_epoch, ra, dec, parallax
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
"""
```

This Hertzsprung-Russell diagram shows the BP-RP color and luminosity of stars in the Gaia catalog.

Selecting stars with `bp-rp` less than 2 excludes many class M dwarf stars, which are low temperature, low luminosity. A star like that at GD-1's distance would be hard to detect, so if it is detected, it it more likely to be in the foreground.

1.12 Cleaning up

Asynchronous jobs have a `jobid`.

```
job1.jobid, job2.jobid
```

```
(None, '1601903242219O')
```

Which you can use to remove the job from the server.

```
Gaia.remove_jobs([job2.jobid])
```

```
Removed jobs: '['1601903242219O']'.
```

If you don't remove it job from the server, it will be removed eventually, so don't feel too bad if you don't clean up after yourself.

1.13 Formatting queries

So far the queries have been string "literals", meaning that the entire string is part of the program. But writing queries yourself can be slow, repetitive, and error-prone.

It is often a good idea to write Python code that assembles a query for you. One useful tool for that is the string `format` method.

As an example, we'll divide the previous query into two parts; a list of column names and a "base" for the query that contains everything except the column names.

Here's the list of columns we'll select.

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

And here's the base; it's a string that contains at least one format specifier in curly brackets (braces).

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

This base query contains one format specifier, `{columns}`, which is a placeholder for the list of column names we will provide.

To assemble the query, we invoke `format` on the base string and provide a keyword argument that assigns a value to `columns`.

```
query3 = query3_base.format(columns=columns)
```

The result is a string with line breaks. If you display it, the line breaks appear as `\n`.

```
query3
```

```
'SELECT TOP 10 \nsource_id, ra, dec, pmra, pmdec, parallax, parallax_error, radial_
↪velocity\nFROM gaiadr2.gaia_source\nWHERE parallax < 1\n  AND bp_rp BETWEEN -0.75␣
↪AND 2\n'
```

But if you print it, the line breaks appear as… line breaks.

```
print(query3)
```

```
SELECT TOP 10
source_id, ra, dec, pmra, pmdec, parallax, parallax_error, radial_velocity
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
```

Notice that the format specifier has been replaced with the value of `columns`.

Let's run it and see if it works:

```
job3 = Gaia.launch_job(query3)
print(job3)
```

```
<Table length=10>
      name       dtype    unit                                description                ␣
↪              n_bad
--------------- ------- -------- ------------------------------------------------------
↪-------------- -----
      source_id   int64          Unique source identifier (unique within a particular␣
↪Data Release)      0
             ra float64      deg                                                        ␣
↪Right ascension     0
            dec float64      deg                                                        ␣
↪  Declination     0
           pmra float64 mas / yr                         Proper motion in right␣
↪ascension direction     0
          pmdec float64 mas / yr                             Proper motion in␣
↪declination direction     0
       parallax float64      mas                                                        ␣
↪     Parallax     0
```

(continues on next page)

(continued from previous page)

```
 parallax_error float64      mas                                                Standard␣
↪error of parallax     0
radial_velocity float64   km / s                                                      ␣
↪Radial velocity    10
Jobid: None
Phase: COMPLETED
Owner: None
Output file: sync_20201005090726.xml.gz
Results: None
```

```
results3 = job3.get_results()
results3
```

```
<Table length=10>
     source_id              ra          ...   parallax_error   radial_velocity
                           deg          ...        mas              km / s
       int64             float64       ...      float64           float64
------------------- ------------------- ... ------------------ ---------------
4467710915011802624  269.9680969307347 ...  0.470352406647465              --
4467706551328679552   270.033164589881 ...  0.927008559859825              --
4467712255037300096  270.7724717923047 ... 0.9719742773203504              --
4467735001181761792  270.3628606248308 ...  0.509812721702093              --
4467737101421916672  270.5110834661444 ... 0.7549581886719651              --
4467707547757327488 269.88746280594927 ... 0.3022057897812064              --
4467732355491087744  270.6730790702491 ... 0.4937921513912002              --
4467717099766944512 270.57667173120825 ... 0.8867339293525688              --
4467719058265781248  270.7248052971514 ...  0.390952370410666              --
4467722326741572352 270.87431291888504 ... 0.1660452431882023              --
```

Good so far.

Exercise: This query always selects sources with `parallax` less than 1. But suppose you want to take that upper bound as an input.

Modify `query3_base` to replace `1` with a format specifier like `{max_parallax}`. Now, when you call `format`, add a keyword argument that assigns a value to `max_parallax`, and confirm that the format specifier gets replaced with the value you provide.

```
# Solution

query4_base = """SELECT TOP 10
{columns}
FROM gaiadr2.gaia_source
WHERE parallax < {max_parallax} AND
bp_rp BETWEEN -0.75 AND 2
"""
```

```
# Solution

query4 = query4_base.format(columns=columns,
                            max_parallax=0.5)
print(query)
```

```
SELECT TOP 10
source_id, ra, dec, pmra, pmdec, parallax, parallax_error, radial_velocity
```

(continues on next page)

(continued from previous page)

```
FROM gaiadr2.gaia_source
WHERE parallax < 0.5 AND
bp_rp BETWEEN -0.75 AND 2
```

Style note: You might notice that the variable names in this notebook are numbered, like `query1`, `query2`, etc.

The advantage of this style is that it isolates each section of the notebook from the others, so if you go back and run the cells out of order, it's less likely that you will get unexpected interactions.

A drawback of this style is that it can be a nuisance to update the notebook if you add, remove, or reorder a section.

What do you think of this choice? Are there alternatives you prefer?

1.14 Summary

This notebook demonstrates the following steps:

1. Making a connection to the Gaia server,
2. Exploring information about the database and the tables it contains,
3. Writing a query and sending it to the server, and finally
4. Downloading the response from the server as an Astropy `Table`.

1.15 Best practices

- If you can't download an entire dataset (or it's not practical) use queries to select the data you need.
- Read the metadata and the documentation to make sure you understand the tables, their columns, and what they mean.
- Develop queries incrementally: start with something simple, test it, and add a little bit at a time.
- Use ADQL features like `TOP` and `COUNT` to test before you run a query that might return a lot of data.
- If you know your query will return fewer than 3000 rows, you can run it synchronously, which might complete faster (but it doesn't seem to make much difference). If it might return more than 3000 rows, you should run it asynchronously.
- ADQL and SQL are not case-sensitive, so you don't have to capitalize the keywords, but you should.
- ADQL and SQL don't require you to break a query into multiple lines, but you should.

Jupyter notebooks can be good for developing and testing code, but they have some drawbacks. In particular, if you run the cells out of order, you might find that variables don't have the values you expect.

There are a few things you can do to mitigate these problems:

- Make each section of the notebook self-contained. Try not to use the same variable name in more than one section.
- Keep notebooks short. Look for places where you can break your analysis into phases with one notebook per phase.

CHAPTER

TWO

CHAPTER 2

This is the second in a series of notebooks related to astronomy data.

As a running example, we are replicating parts of the analysis in a recent paper, "Off the beaten path: Gaia reveals GD-1 stars outside of the main stream" by Adrian M. Price-Whelan and Ana Bonaca.

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

In this notebook, we'll pick up where we left off and write a query to select stars from the region of the sky where we expect GD-1 to be.

2.1 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 the `Gala` library 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.

2.2 Installing libraries

If you are running this notebook on Colab, you can run the following cell to install Astroquery and a the other libraries we'll use.

If you are running this notebook on your own computer, you might have to install these libraries yourself.

If you are using this notebook as part of a Carpentries workshop, you should have received setup instructions.

TODO: Add a link to the instructions.

```
# If we're running on Colab, install libraries

import sys
IN_COLAB = 'google.colab' in sys.modules

if IN_COLAB:
    !pip install astroquery astro-gala pyia
```

2.3 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 "all the objects ... in a circular region centered at (266.41683, -29.00781) with a search radius of 5 arcmin (0.08333 deg)."

```
query = """
SELECT
TOP 10 source_id
FROM gaiadr2.gaia_source
WHERE 1=CONTAINS(
  POINT(ra, dec),
  CIRCLE(266.41683, -29.00781, 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)
result = job.get_results()
result
```

```
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
```

```
<Table length=10>
     source_id
       int64
------------------
4057468321929794432
4057468287575835392
4057482027171038976
4057470349160630656
4057470039924301696
4057469868125641984
4057468351995073024
4057469661959554560
4057470520960672640
4057470555320409600
```

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?

2.4 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:

Along the axis of right ascension (ϕ_1) the figure extends from -100 to 20 degrees.

Along the axis of declination (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 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, from -55 to -45 degrees right ascension and -8 to 4 degrees of declination.

But first we let's see how to represent quantities with units like degrees.

2.5 Working with coordinates

Coordinates 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 disastrous 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
```

```
<module 'astropy.units' from '/home/downey/anaconda3/envs/AstronomicalData/lib/
↪python3.8/site-packages/astropy/units/__init__.py'>
```

`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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
'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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 'Tdeg',
 'Tdyn',
 'TeV',
 'Terg',
 'Tesla',
 'Tg',
 'Th',
 'TiB',
 'Tib',
 'Tibit',
 'Tibyte',
 'Tk',
 'Tl',
 'Tlm',
 'Tlx',
 'Tlyr',
 'Tm',
 'Tmag',
 'Tmin',
 'Tmol',
 'Tohm',
 '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',
```

(continues on next page)

(continued from previous page)

```
'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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
```

(continues on next page)

(continued from previous page)

```
 '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',
 'dT',
 ...]
```

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

```
coord = 30 * u.deg
type(coord)
```

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

The result is a `Quantity` object.

Jupyter knows how to display `Quantities` like this:

```
coord
```

$30\ ^\circ$

2.6 Selecting a rectangle

Now we'll select a rectangle from -55 to -45 degrees right ascension and -8 to 4 degrees of declination.

We'll define variables to contain these limits.

```
phi1_min = -55
phi1_max = -45
phi2_min = -8
phi2_max = 4
```

To represent a rectangle, we'll use two lists of coordinates and multiply by their units.

```
phi1_rect = [phi1_min, phi1_min, phi1_max, phi1_max] * u.deg
phi2_rect = [phi2_min, phi2_max, phi2_max, phi2_min] * u.deg
```

`phi1_rect` and `phi2_rect` represent the coordinates of the corners of a rectangle.

But they are in "a Heliocentric spherical coordinate system defined by the orbit of the GD1 stream"

In order to use them in a Gaia query, we have to convert them to International Celestial Reference System (ICRS) coordinates. We can do that by storing the coordinates in a `GD1Koposov10` object provided by Gala.

```
import gala.coordinates as gc

corners = gc.GD1Koposov10(phi1=phi1_rect, phi2=phi2_rect)
type(corners)
```

```
gala.coordinates.gd1.GD1Koposov10
```

We can display the result like this:

```
corners
```

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

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

```
import astropy.coordinates as coord

corners_icrs = corners.transform_to(coord.ICRS)
type(corners_icrs)
```

```
astropy.coordinates.builtin_frames.icrs.ICRS
```

The result is an `ICRS` object.

```
corners_icrs
```

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

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

2.7 Selecting 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)
"""
```

`corners_icrs` behaves like a list, so we can use a `for` loop to iterate through the points.

```
for point in corners_icrs:
    print(point)
```

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

From that, we can select the coordinates `ra` and `dec`:

```
for point in corners_icrs:
    print(point.ra, point.dec)
```

```
146d16m31.1993s 19d15m42.8754s
135d25m17.902s 25d52m38.594s
141d36m09.5337s 34d18m17.3891s
152d49m00.1576s 27d08m10.0051s
```

The results are quantities with units, but if we select the `value` part, we get a dimensionless floating-point number.

```
for point in corners_icrs:
    print(point.ra.value, point.dec.value)
```

```
146.27533313607782 19.261909820533692
135.42163944306296 25.87738722767213
141.60264825107333 34.304830296257144
152.81671044675923 27.136112541397996
```

We can use string `format` to convert these numbers to strings.

```
point_base = "{point.ra.value}, {point.dec.value}"

t = [point_base.format(point=point)
     for point in corners_icrs]
t
```

```
['146.27533313607782, 19.261909820533692',
 '135.42163944306296, 25.87738722767213',
 '141.60264825107333, 34.304830296257144',
 '152.81671044675923, 27.136112541397996']
```

The result is a list of strings, which we can join into a single string using `join`.

```
point_list = ', '.join(t)
point_list
```

```
'146.27533313607782, 19.261909820533692, 135.42163944306296, 25.87738722767213, 141.
↪60264825107333, 34.304830296257144, 152.81671044675923, 27.136112541397996'
```

Notice that we invoke `join` on a string and pass the list as an argument.

Before we can assemble the query, we need `columns` again (as we saw in the previous notebook).

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

Here's the base for the query, with format specifiers for `columns` and `point_list`.

```
query_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}))
"""
```

And here's the result:

```
query = query_base.format(columns=columns,
                          point_list=point_list)
print(query)
```

```
SELECT source_id, ra, dec, pmra, pmdec, parallax, parallax_error, radial_velocity
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
  AND 1 = CONTAINS(POINT(ra, dec),
                   POLYGON(146.27533313607782, 19.261909820533692, 135.42163944306296,
↪ 25.87738722767213, 141.60264825107333, 34.304830296257144, 152.81671044675923, 27.
↪136112541397996))
```

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

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

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

```
INFO: Query finished. [astroquery.utils.tap.core]
<Table length=140340>
      name       dtype    unit                                  description
↪              n_bad
--------------- ------- -------- ------------------------------------------------------------
↪------------- ------
      source_id   int64          Unique source identifier (unique within a particular␣
↪Data Release)       0
             ra float64      deg                                                            ␣
↪Right ascension      0
            dec float64      deg                                                            ␣
↪  Declination      0
           pmra float64 mas / yr                        Proper motion in right␣
↪ascension direction      0
          pmdec float64 mas / yr                            Proper motion in␣
↪declination direction      0
       parallax float64      mas                                                            ␣
↪     Parallax      0
 parallax_error float64      mas                                          Standard␣
↪error of parallax      0
radial_velocity float64   km / s                                                            ␣
↪Radial velocity 139374
Jobid: 16031149806580
Phase: COMPLETED
Owner: None
Output file: async_20201019094300.vot
Results: None
```

Here are the results.

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

```
140340
```

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

2.8 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.

To see how big the file is, we can use `ls` with the `-lh` option, which prints information about the file including its size in human-readable form.

```
!ls -lh gd1_results.fits
```

```
-rw-rw-r-- 1 downey downey 8.6M Oct 19 09:43 gd1_results.fits
```

The file is about 8.6 MB. If you are using Windows, `ls` might not work; in that case, try:

```
!dir gd1_results.fits
```

2.9 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.

2.10 Best practices

- For measurements with units, use `Quantity` objects that represent units explicitly and check for errors.
- Use the `format` function to compose queries; it is often faster 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.

CHAPTER

THREE

CHAPTER 3

This is the third in a series of notebooks related to astronomy data.

As a running example, we are replicating parts of the analysis in a recent paper, "Off the beaten path: Gaia reveals GD-1 stars outside of the main stream" by Adrian M. Price-Whelan and Ana Bonaca.

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

In the second lesson, we wrote a query to select stars from the region of the sky where we expect GD-1 to be, and saved the results in a FITS file.

Now we'll read that data back and implement the next step in the analysis, identifying stars with the proper motion we expect for GD-1.

3.1 Outline

Here are the steps in this lesson:

1. We'll read back the results from the previous lesson, which we saved in a FITS file.
2. Then we'll transform the coordinates and proper motion data from ICRS back to the coordinate frame of GD-1.
3. We'll put those results into a Pandas `DataFrame`, which we'll use to select stars near the centerline of GD-1.
4. Plotting the proper motion of those stars, we'll identify a region of proper motion for stars that are likely to be in GD-1.
5. Finally, we'll select and plot the stars whose proper motion is in that region.

After completing this lesson, you should be able to

- Select rows and columns from an Astropy `Table`.
- Use Matplotlib to make a scatter plot.
- Use Gala to transform coordinates.
- Make a Pandas `DataFrame` and use a Boolean `Series` to select rows.
- Save a `DataFrame` in an HDF5 file.

3.2 Installing libraries

If you are running this notebook on Colab, you can run the following cell to install Astroquery and a the other libraries we'll use.

If you are running this notebook on your own computer, you might have to install these libraries yourself.

If you are using this notebook as part of a Carpentries workshop, you should have received setup instructions.

TODO: Add a link to the instructions.

```
# If we're running on Colab, install libraries

import sys
IN_COLAB = 'google.colab' in sys.modules

if IN_COLAB:
    !pip install astroquery astro-gala pyia python-wget
```

3.3 Reload the data

In the previous lesson, we ran a query on the Gaia server and downloaded data for roughly 100,000 stars. We saved the data in a FITS file so that now, picking up where we left off, we can read the data from a local file rather than running the query again.

If you ran the previous lesson successfully, you should already have a file called `gd1_results.fits` that contains the data we downloaded.

If not, you can run the following cell, which downloads the data from our repository.

```
import os
from wget import download

filename = 'gd1_results.fits'
path = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(path+filename))
```

Now here's how we can read the data from the file back into an Astropy `Table`:

```
from astropy.table import Table

results = Table.read(filename)
```

The result is an Astropy `Table`.

We can use `info` to refresh our memory of the contents.

```
results.info
```

```
<Table length=140340>
      name       dtype    unit                                description
--------------- ------- -------- ------------------------------------------------------
 ↪-------------
      source_id   int64          Unique source identifier (unique within a particular␣
↪Data Release)
```

(continues on next page)

(continued from previous page)

```
            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
      parallax float64      mas                                                 ␣
↪     Parallax
 parallax_error float64      mas                                      Standard␣
↪error of parallax
radial_velocity float64   km / s                                                ␣
↪Radial velocity
```

3.4 Selecting rows and columns

In this section we'll see operations for selecting columns and rows from an Astropy `Table`. You can find more information about these operations in the Astropy documentation.

We can get the names of the columns like this:

```
results.colnames
```

```
['source_id',
 'ra',
 'dec',
 'pmra',
 'pmdec',
 'parallax',
 'parallax_error',
 'radial_velocity']
```

And select an individual column like this:

```
results['ra']
```

```
<Column name='ra' dtype='float64' unit='deg' description='Right ascension'␣
↪length=140340>
142.48301935991023
142.25452941346344
142.64528557468074
142.57739430926034
142.58913564478618
141.81762228999614
143.18339801317677
 142.9347319464589
142.26769745823267
142.89551292869012
 142.2780935768316
142.06138786534987
               ...
143.05456487172972
```

(continues on next page)

(continued from previous page)

```
 144.0436496516182
144.06566578919313
144.13177563215973
143.77696341662764
  142.945956347594
142.97282480557786
 143.4166017695258
143.64484588686904
143.41554585481808
 143.6908739159247
 143.7702681295401
```

The result is a `Column` object that contains the data, and also the data type, units, and name of the column.

```
type(results['ra'])
```

```
astropy.table.column.Column
```

The rows in the `Table` are numbered from 0 to `n-1`, where `n` is the number of rows. We can select the first row like this:

```
results[0]
```

```
<Row index=0>
    source_id              ra                 dec                 pmra              ␣
↪pmdec              parallax        parallax_error  radial_velocity
                         deg                deg               mas / yr            mas /
↪ yr             mas                mas              km / s
      int64             float64            float64            float64             ␣
↪float64            float64           float64          float64
------------------ ----------------- ----------------- ------------------ ----------
↪------- ------------------ ----------------- ---------------
637987125186749568 142.48301935991023 21.75771616932985 -2.5168384683875766 2.
↪941813096629439 -0.2573448962333354 0.823720794509811           1e+20
```

As you might have guessed, the result is a `Row` object.

```
type(results[0])
```

```
astropy.table.row.Row
```

Notice that the bracket operator selects both columns and rows. You might wonder how it knows which to select.

If the expression in brackets is a string, it selects a column; if the expression is an integer, it selects a row.

If you apply the bracket operator twice, you can select a column and then an element from the column.

```
results['ra'][0]
```

```
142.48301935991023
```

Or you can select a row and then an element from the row.

```
results[0]['ra']
```

```
142.48301935991023
```

You get the same result either way.

3.5 Scatter plot

To see what the results look like, we'll use a scatter plot. The library we'll use is Matplotlib, which is the most widely-used plotting library for Python.

The Matplotlib interface is based on MATLAB (hence the name), so if you know MATLAB, some of it will be familiar.

We'll import like this.

```
import matplotlib.pyplot as plt
```

Pyplot part of the Matplotlib library. It is conventional to import it using the shortened name `plt`.

Pyplot provides two functions that can make scatterplots, plt.scatter and plt.plot.

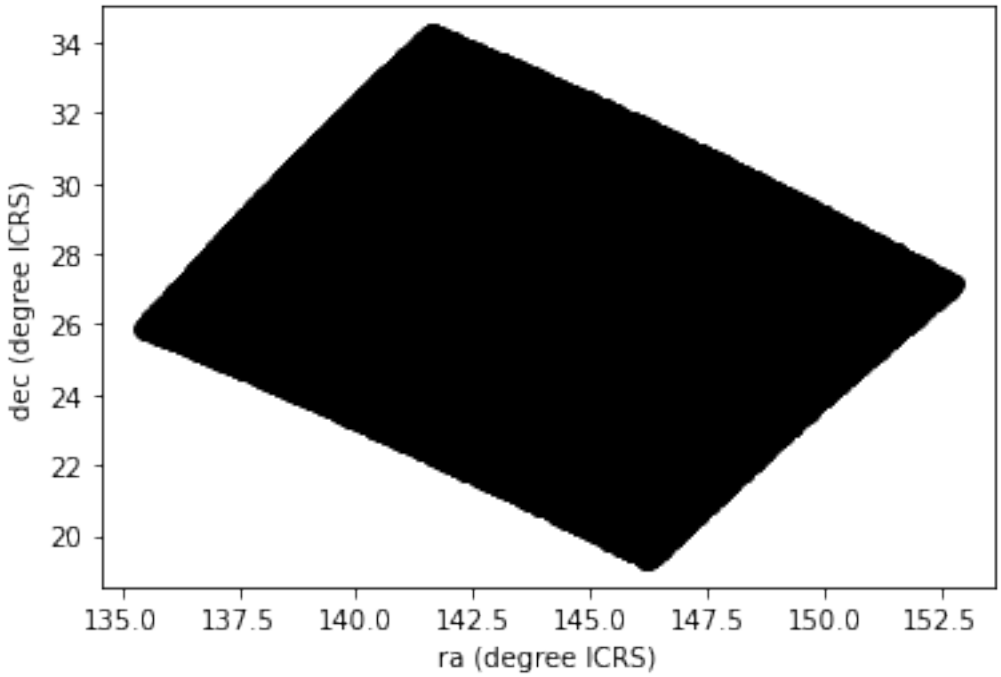
- `scatter` is more versatile; for example, you can make every point in a scatter plot a different color.
- `plot` is more limited, but for simple cases, it can be substantially faster.

Jake Vanderplas explains these differences in The Python Data Science Handbook

Since we are plotting more than 100,000 points and they are all the same size and color, we'll use `plot`.

Here's a scatter plot with right ascension on the x-axis and declination on the y-axis, both ICRS coordinates in degrees.

```
x = results['ra']
y = results['dec']
plt.plot(x, y, 'ko')

plt.xlabel('ra (degree ICRS)')
plt.ylabel('dec (degree ICRS)');
```



The arguments to `plt.plot` are `x`, `y`, and a string that specifies the style. In this case, the letters `ko` indicate that we want a black, round marker (`k` is for black because `b` is for blue).

The functions `xlabel` and `ylabel` put labels on the axes.

This scatter plot has a problem. It is "overplotted", which means that there are so many overlapping points, we can't distinguish between high and low density areas.

To fix this, we can provide optional arguments to control the size and transparency of the points.

Exercise: In the call to `plt.plot`, add the keyword argument `markersize=0.1` to make the markers smaller.

Then add the argument `alpha=0.1` to make the markers nearly transparent.

Adjust these arguments until you think the figure shows the data most clearly.

Note: Once you have made these changes, you might notice that the figure shows stripes with lower density of stars. These stripes are caused by the way Gaia scans the sky, which you can read about here. The dataset we are using, Gaia Data Release 2, covers 22 months of observations; during this time, some parts of the sky were scanned more than others.

3.6 Transform back

Remember that we selected data from a rectangle of coordinates in the `GD1Koposov10` frame, then transformed them to ICRS when we constructed the query. The coordinates in `results` are in ICRS.

To plot them, we will transform them back to the `GD1Koposov10` frame; that way, the axes of the figure are aligned with the GD-1, which will make it easy to select stars near the centerline of the stream.

To do that, we'll put the results into a `GaiaData` object, provided by the pyia library.

```
from pyia import GaiaData
```

(continues on next page)

(continued from previous page)

```
gaia_data = GaiaData(results)
type(gaia_data)
```

```
pyia.data.GaiaData
```

Now we can extract sky coordinates from the `GaiaData` object, like this:

```
import astropy.units as u

skycoord = gaia_data.get_skycoord(
                distance=8*u.kpc,
                radial_velocity=0*u.km/u.s)
```

We provide `distance` and `radial_velocity` to prepare the data for reflex correction, which we explain below.

```
type(skycoord)
```

```
astropy.coordinates.sky_coordinate.SkyCoord
```

The result is an Astropy `SkyCoord` object (documentation here), which provides `transform_to`, so we can transform the coordinates to other frames.

```
import gala.coordinates as gc

transformed = skycoord.transform_to(gc.GD1Koposov10)
type(transformed)
```

```
astropy.coordinates.sky_coordinate.SkyCoord
```

The result is another `SkyCoord` object, now in the `GD1Koposov10` frame.

The next step is to correct the proper motion measurements from Gaia for reflex due to the motion of our solar system around the Galactic center.

When we created `skycoord`, we provided `distance` and `radial_velocity` as arguments, which means we ignore the measurements provided by Gaia and replace them with these fixed values.

That might seem like a strange thing to do, but here's the motivation:

- Because the stars in GD-1 are so far away, the distance estimates we get from Gaia, which are based on parallax, are not very precise. So we replace them with our current best estimate of the mean distance to GD-1, about 8 kpc. See Koposov, Rix, and Hogg, 2010.
- For the other stars in the table, this distance estimate will be inaccurate, so reflex correction will not be correct. But that should have only a small effect on our ability to identify stars with the proper motion we expect for GD-1.
- The measurement of radial velocity has no effect on the correction for proper motion; the value we provide is arbitrary, but we have to provide a value to avoid errors in the reflex correction calculation.

We are grateful to Adrian Price-Whelen for his help explaining this step in the analysis.

With this preparation, we can use `reflex_correct` from Gala (documentation here) to correct for solar reflex motion.

```
gd1_coord = gc.reflex_correct(transformed)

type(gd1_coord)
```

```
astropy.coordinates.sky_coordinate.SkyCoord
```

The result is a `SkyCoord` object that contains

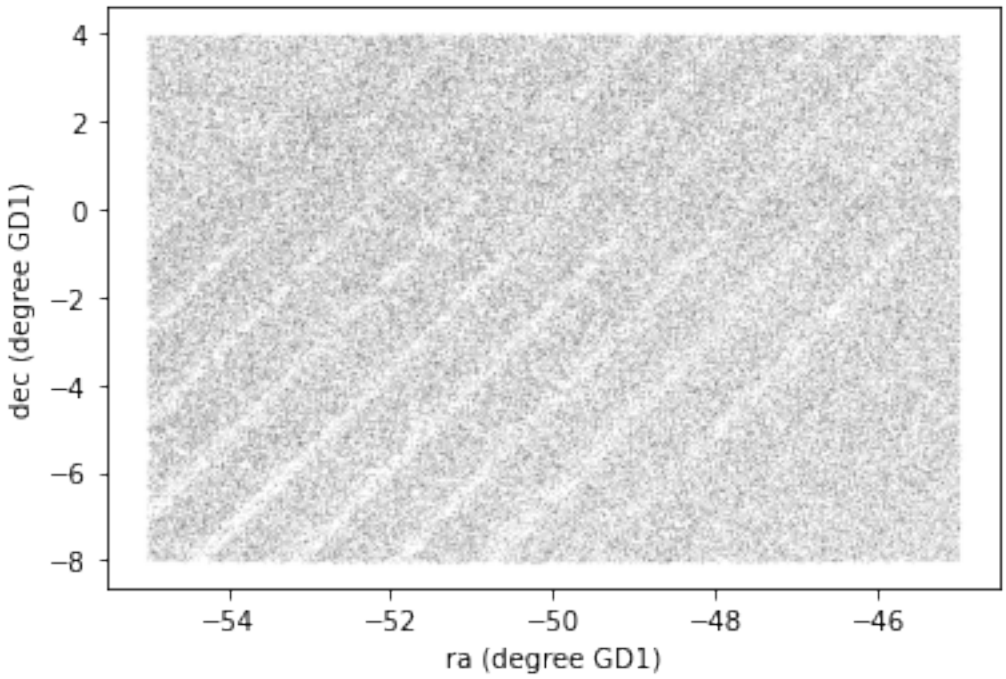
- The transformed coordinates as attributes named `phi1` and `phi2`, which represent right ascension and declination in the `GD1Koposov10` frame.
- The transformed and corrected proper motions as `pm_phi1_cosphi2` and `pm_phi2`.

We can select the coordinates like this:

```
phi1 = gd1_coord.phi1
phi2 = gd1_coord.phi2
```

And plot them like this:

```
plt.plot(phi1, phi2, 'ko', markersize=0.1, alpha=0.2)

plt.xlabel('ra (degree GD1)')
plt.ylabel('dec (degree GD1)');
```



Remember that we started with a rectangle in GD-1 coordinates. When transformed to ICRS, it's a non-rectangular polygon. Now that we have transformed back to GD-1 coordinates, it's a rectangle again.

3.7 Pandas DataFrame

At this point we have three objects containing different subsets of the data.

```
type(results)
```

```
astropy.table.table.Table
```

```
type(gaia_data)
```

```
pyia.data.GaiaData
```

```
type(gd1_coord)
```

```
astropy.coordinates.sky_coordinate.SkyCoord
```

On one hand, this makes sense, since each object provides different capabilities. But working with three different object types can be awkward.

It will be more convenient to choose one object and get all of the data into it. We'll use a Pandas DataFrame, for two reasons:

1. It provides capabilities that are pretty much a superset of the other data structures, so it's the all-in-one solution.
2. Pandas is a general-purpose tool that is useful in many domains, especially data science. If you are going to develop expertise in one tool, Pandas is a good choice.

However, compared to an Astropy `Table`, Pandas has one big drawback: it does not keep the metadata associated with the table, including the units for the columns.

It's easy to convert a `Table` to a Pandas `DataFrame`.

```
import pandas as pd

df = results.to_pandas()
df.shape
```

```
(140340, 8)
```

`DataFrame` provides `shape`, which shows the number of rows and columns.

It also provides `head`, which displays the first few rows. It is useful for spot-checking large results as you go along.

```
df.head()
```

```
            source_id          ra        dec       pmra      pmdec  parallax  \
0  637987125186749568  142.483019  21.757716  -2.516838   2.941813 -0.257345
1  638285195917112960  142.254529  22.476168   2.662702 -12.165984  0.422728
2  638073505568978688  142.645286  22.166932  18.306747  -7.950660  0.103640
3  638086386175786752  142.577394  22.227920   0.987786  -2.584105 -0.857327
4  638049655615392384  142.589136  22.110783   0.244439  -4.941079  0.099625

   parallax_error  radial_velocity
0        0.823721     1.000000e+20
1        0.297472     1.000000e+20
2        0.544584     1.000000e+20
```

(continues on next page)

(continued from previous page)

```
3         1.059607     1.000000e+20
4         0.486224     1.000000e+20
```

Python detail: `shape` is an attribute, so we can display it's value without calling it as a function; `head` is a function, so we need the parentheses.

Now we can extract the columns we want from `gd1_coord` and add them as columns in the `DataFrame`. `phi1` and `phi2` contain the transformed coordinates.

```
df['phi1'] = gd1_coord.phi1
df['phi2'] = gd1_coord.phi2
df.shape
```

```
(140340, 10)
```

`pm_phi1_cosphi2` and `pm_phi2` contain the components of proper motion in the transformed frame.

```
df['pm_phi1'] = gd1_coord.pm_phi1_cosphi2
df['pm_phi2'] = gd1_coord.pm_phi2
df.shape
```

```
(140340, 12)
```

Detail: If you notice that `SkyCoord` has an attribute called `proper_motion`, you might wonder why we are not using it.

We could have: `proper_motion` contains the same data as `pm_phi1_cosphi2` and `pm_phi2`, but in a different format.

3.8 Plot proper motion

Now we are ready to replicate one of the panels in Figure 1 of the Price-Whelan and Bonaca paper, the one that shows the components of proper motion as a scatter plot:

In this figure, the shaded area is a high-density region of stars with the proper motion we expect for stars in GD-1.

- Due to the nature of tidal streams, we expect the proper motion for most stars to be along the axis of the stream; that is, we expect motion in the direction of `phi2` to be near 0.
- In the direction of `phi1`, we don't have a prior expectation for proper motion, except that it should form a cluster at a non-zero value.

To locate this cluster, we'll select stars near the centerline of GD-1 and plot their proper motion.

3.9 Selecting the centerline

As we can see in the following figure, many stars in GD-1 are less than 1 degree of declination from the line `phi2=0`.

If we select stars near this line, they are more likely to be in GD-1.

We'll start by selecting the `phi2` column from the `DataFrame`:

```
phi2 = df['phi2']
type(phi2)
```

```
pandas.core.series.Series
```

The result is a `Series`, which is the structure Pandas uses to represent columns.

We can use a comparison operator, `>`, to compare the values in a `Series` to a constant.

```
phi2_min = -1.0 * u.deg
phi2_max = 1.0 * u.deg

mask = (df['phi2'] > phi2_min)
type(mask)
```

```
pandas.core.series.Series
```

```
mask.dtype
```

```
dtype('bool')
```

The result is a `Series` of Boolean values, that is, `True` and `False`.

```
mask.head()
```

```
0    False
1    False
2    False
3    False
4    False
Name: phi2, dtype: bool
```

A Boolean `Series` is sometimes called a "mask" because we can use it to mask out some of the rows in a `DataFrame` and select the rest, like this:

```
selected = df[mask]
type(selected)
```

```
pandas.core.frame.DataFrame
```

`selected` is a `DataFrame` that contains only the rows from `df` that correspond to `True` values in `mask`.

The previous mask selects all stars where `phi2` exceeds `phi2_min`; now we'll select stars where `phi2` falls between `phi2_min` and `phi2_max`.

```
phi_mask = ((df['phi2'] > phi2_min) &
            (df['phi2'] < phi2_max))
```

The `&` operator computes "logical AND", which means the result is true where elements from both Boolean `Series` are true.

The sum of a Boolean `Series` is the number of `True` values, so we can use `sum` to see how many stars are in the selected region.

```
phi_mask.sum()
```

```
25084
```

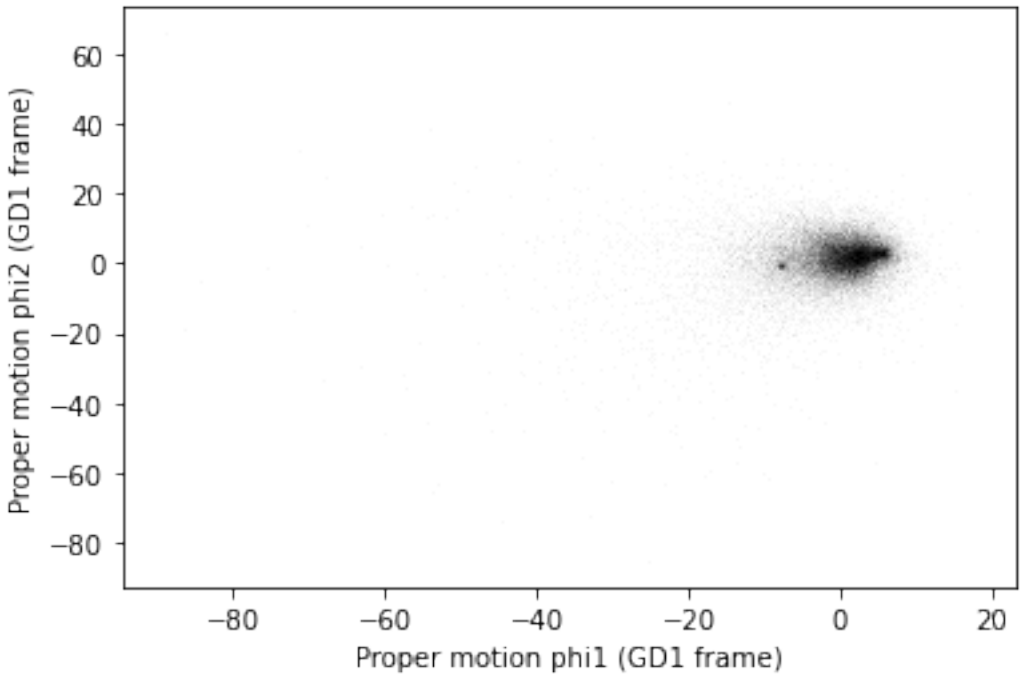
And we can use `phi1_mask` to select stars near the centerline, which are more likely to be in GD-1.

```
centerline = df[phi_mask]
len(centerline)
```

```
25084
```

Here's a scatter plot of proper motion for the selected stars.

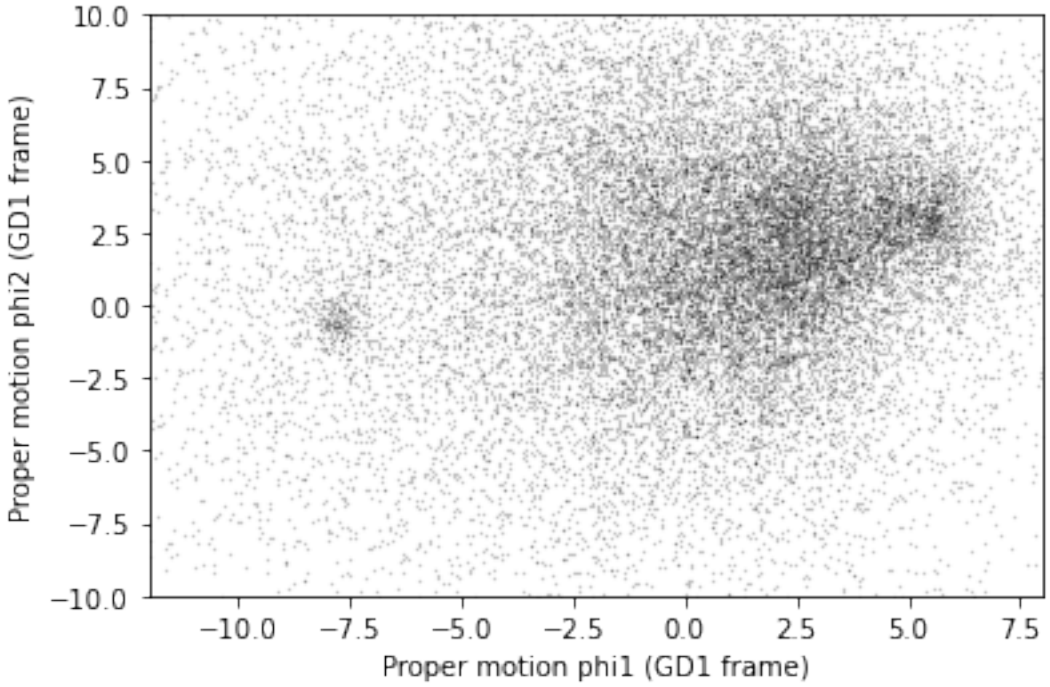
```
pm1 = centerline['pm_phi1']
pm2 = centerline['pm_phi2']

plt.plot(pm1, pm2, 'ko', markersize=0.1, alpha=0.1)

plt.xlabel('Proper motion phi1 (GD1 frame)')
plt.ylabel('Proper motion phi2 (GD1 frame)');
```



Looking at these results, we see a large cluster around (0, 0), and a smaller cluster near (0, -10).

We can use `xlim` and `ylim` to set the limits on the axes and zoom in on the region near (0, 0).

```
pm1 = centerline['pm_phi1']
pm2 = centerline['pm_phi2']

plt.plot(pm1, pm2, 'ko', markersize=0.3, alpha=0.3)

plt.xlabel('Proper motion phi1 (GD1 frame)')
plt.ylabel('Proper motion phi2 (GD1 frame)')

plt.xlim(-12, 8)
plt.ylim(-10, 10);
```



Now we can see the smaller cluster more clearly.

You might notice that our figure is less dense than the one in the paper. That's because we started with a set of stars from a relatively small region. The figure in the paper is based on a region about 10 times bigger.

In the next lesson we'll go back and select stars from a larger region. But first we'll use the proper motion data to identify stars likely to be in GD-1.

3.10 Filtering based on proper motion

The next step is to select stars in the "overdense" region of proper motion, which are candidates to be in GD-1.

In the original paper, Price-Whelan and Bonaca used a polygon to cover this region, as shown in this figure.

We'll use a simple rectangle for now, but in a later lesson we'll see how to select a polygonal region as well.

Here are bounds on proper motion we chose by eye,

```
pm1_min = -8.9
pm1_max = -6.9
pm2_min = -2.2
pm2_max =  1.0
```

To draw these bounds, we'll make two lists containing the coordinates of the corners of the rectangle.

```
pm1_rect = [pm1_min, pm1_min, pm1_max, pm1_max, pm1_min] * u.mas/u.yr
pm2_rect = [pm2_min, pm2_max, pm2_max, pm2_min, pm2_min] * u.mas/u.yr
```

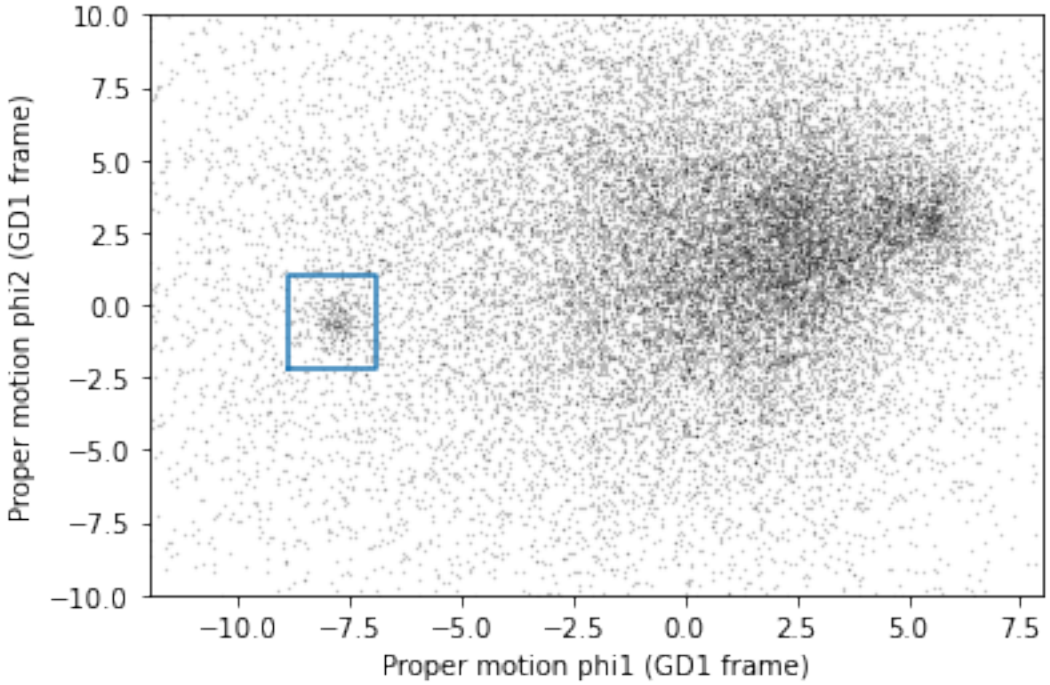
Here's what the plot looks like with the bounds we chose.

```
plt.plot(pm1, pm2, 'ko', markersize=0.3, alpha=0.3)
plt.plot(pm1_rect, pm2_rect, '-')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Proper motion phi1 (GD1 frame)')
plt.ylabel('Proper motion phi2 (GD1 frame)')

plt.xlim(-12, 8)
plt.ylim(-10, 10);
```



To select rows that fall within these bounds, we'll use the following function, which uses Pandas operators to make a mask that selects rows where `series` falls between `low` and `high`.

```
def between(series, low, high):
    """Make a Boolean Series.

    series: Pandas Series
    low: lower bound
    high: upper bound

    returns: Boolean Series
    """
    return (series > low) & (series < high)
```

The following mask select stars with proper motion in the region we chose.

```
pm_mask = (between(df['pm_phi1'], pm1_min, pm1_max) &
           between(df['pm_phi2'], pm2_min, pm2_max))
```

Again, the sum of a Boolean series is the number of `True` values.

```
pm_mask.sum()
```

```
1049
```

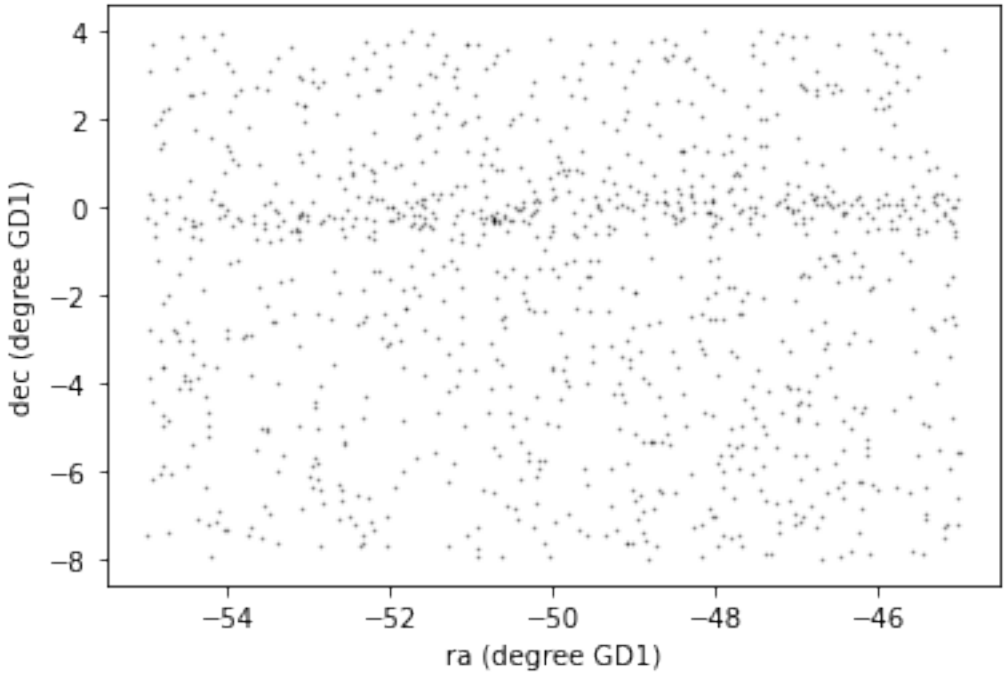
Now we can use this mask to select rows from `df`.

```
selected = df[pm_mask]
len(selected)
```

```
1049
```

These are the stars we think are likely to be in GD-1. Let's see what they look like, plotting their coordinates (not their proper motion).

```
phi1 = selected['phi1']
phi2 = selected['phi2']

plt.plot(phi1, phi2, 'ko', markersize=0.5, alpha=0.5)

plt.xlabel('ra (degree GD1)')
plt.ylabel('dec (degree GD1)');
```



Now that's starting to look like a tidal stream!

3.11 Saving the DataFrame

At this point we have run a successful query and cleaned up the results; this is a good time to save the data.

To save a Pandas `DataFrame`, one option is to convert it to an Astropy `Table`, like this:

```
selected_table = Table.from_pandas(selected)
type(selected_table)
```

```
astropy.table.table.Table
```

Then we could write the `Table` to a FITS file, as we did in the previous lesson.

But Pandas provides functions to write DataFrames in other formats; to see what they are find the functions here that begin with `to_`.

One of the best options is HDF5, which is Version 5 of Hierarchical Data Format.

HDF5 is a binary format, so files are small and fast to read and write (like FITS, but unlike XML).

An HDF5 file is similar to an SQL database in the sense that it can contain more than one table, although in HDF5 vocabulary, a table is called a Dataset. (Multi-extension FITS files can also contain more than one table.)

And HDF5 stores the metadata associated with the table, including column names, row labels, and data types (like FITS).

Finally, HDF5 is a cross-language standard, so if you write an HDF5 file with Pandas, you can read it back with many other software tools (more than FITS).

Before we write the HDF5, let's delete the old one, if it exists.

```
!rm -f gd1_dataframe.hdf5
```

We can write a Pandas `DataFrame` to an HDF5 file like this:

```
filename = 'gd1_dataframe.hdf5'

df.to_hdf(filename, 'df')
```

Because an HDF5 file can contain more than one Dataset, we have to provide a name, or "key", that identifies the Dataset in the file.

We could use any string as the key, but in this example I use the variable name `df`.

Exercise: We're going to need `centerline` and `selected` later as well. Write a line or two of code to add it as a second Dataset in the HDF5 file.

```
# Solution

centerline.to_hdf(filename, 'centerline')
selected.to_hdf(filename, 'selected')
```

Detail: Reading and writing HDF5 tables requires a library called `PyTables` that is not always installed with Pandas. You can install it with pip like this:

```
pip install tables
```

If you install it using Conda, the name of the package is `pytables`.

```
conda install pytables
```

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

```
!ls -lh gd1_dataframe.hdf5
```

```
-rw-rw-r-- 1 downey downey 17M Oct 19 12:05 gd1_dataframe.hdf5
```

If you are using Windows, `ls` might not work; in that case, try:

```
!dir gd1_dataframe.hdf5
```

We can read the file back like this:

```
read_back_df = pd.read_hdf(filename, 'df')
read_back_df.shape
```

```
(140340, 12)
```

Pandas can write a variety of other formats, which you can read about here.

3.12 Summary

In this lesson, we re-loaded the Gaia data we saved from a previous query.

We transformed the coordinates and proper motion from ICRS to a frame aligned with GD-1, and stored the results in a Pandas `DataFrame`.

Then we replicated the selection process from the Price-Whelan and Bonaca paper:

- We selected stars near the centerline of GD-1 and made a scatter plot of their proper motion.
- We identified a region of proper motion that contains stars likely to be in GD-1.
- We used a Boolean `Series` as a mask to select stars whose proper motion is in that region.

So far, we have used data from a relatively small region of the sky. In the next lesson, we'll write a query that selects stars based on proper motion, which will allow us to explore a larger region.

3.13 Best practices

- When you make a scatter plot, adjust the size of the markers and their transparency so the figure is not overplotted; otherwise it can misrepresent the data badly.
- For simple scatter plots in Matplotlib, `plot` is faster than `scatter`.
- An Astropy `Table` and a Pandas `DataFrame` are similar in many ways and they provide many of the same functions. They have pros and cons, but for many projects, either one would be a reasonable choice.

CHAPTER

FOUR

CHAPTER 4

This is the fourth in a series of notebooks related to astronomy data.

As a running example, we are replicating parts of the analysis in a recent paper, "Off the beaten path: Gaia reveals GD-1 stars outside of the main stream" by Adrian M. Price-Whelan and Ana Bonaca.

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

In the second lesson, we write a query to select stars from the region of the sky where we expect GD-1 to be, and save the results in a FITS file.

In the third lesson, we read that data back and identified stars with the proper motion we expect for GD-1.

4.1 Outline

Here are the steps in this lesson:

1. Using data from the previous lesson, we'll identify the values of proper motion for stars likely to be in GD-1.
2. Then we'll compose an ADQL query that selects stars based on proper motion, so we can download only the data we need.
3. We'll also see how to write the results to a CSV file.

That will make it possible to search a bigger region of the sky in a single query.

After completing this lesson, you should be able to

- Convert proper motion between frames.
- Write an ADQL query that selects based on proper motion.

4.2 Installing libraries

If you are running this notebook on Colab, you can run the following cell to install Astroquery and a the other libraries we'll use.

If you are running this notebook on your own computer, you might have to install these libraries yourself.

If you are using this notebook as part of a Carpentries workshop, you should have received setup instructions.

TODO: Add a link to the instructions.

```
# If we're running on Colab, install libraries

import sys
IN_COLAB = 'google.colab' in sys.modules

if IN_COLAB:
    !pip install astroquery astro-gala pyia python-wget
```

4.3 Reload the data

The following cells download the data from the previous lesson, if necessary, and load it into a Pandas `DataFrame`.

```
import os
from wget import download

filename = 'gd1_dataframe.hdf5'
path = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(path+filename))
```

```
import pandas as pd

df = pd.read_hdf(filename, 'df')
centerline = pd.read_hdf(filename, 'centerline')
selected = pd.read_hdf(filename, 'selected')
```

4.4 Selection by proper motion

At this point we have downloaded data for a relatively large number of stars (more than 100,000) and selected a relatively small number (around 1000).

It would be more efficient to use ADQL to select only the stars we need. That would also make it possible to download data covering a larger region of the sky.

However, the selection we did was based on proper motion in the `GD1Koposov10` frame. In order to do the same selection in ADQL, we have to work with proper motions in ICRS.

As a reminder, here's the rectangle we selected based on proper motion in the `GD1Koposov10` frame.

```
pm1_min = -8.9
pm1_max = -6.9
pm2_min = -2.2
pm2_max =  1.0
```

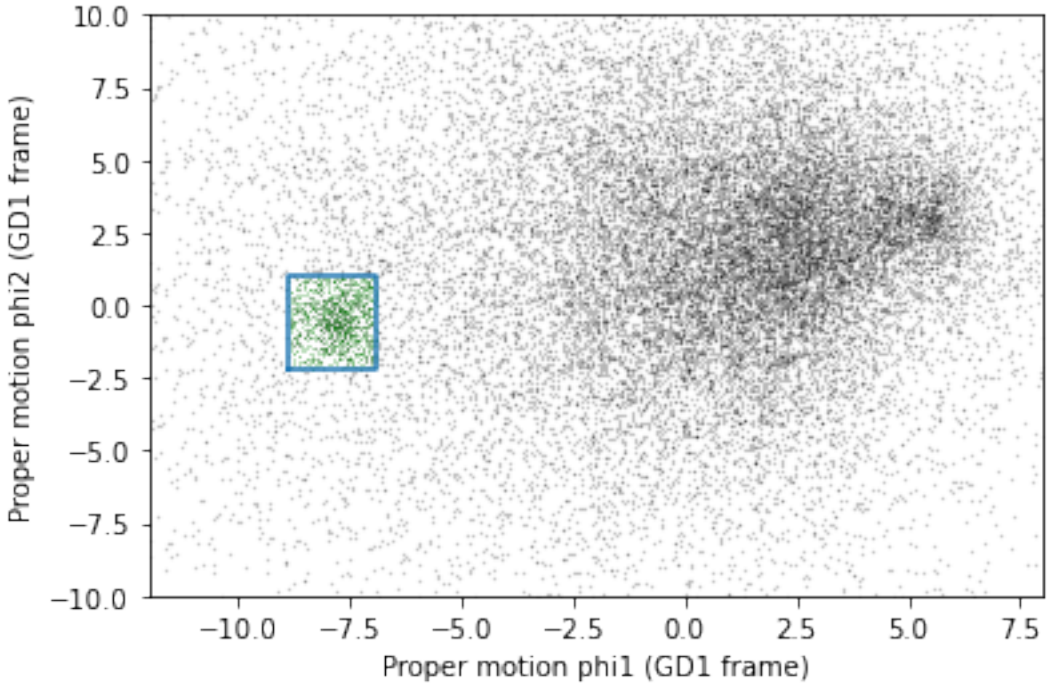
```
import astropy.units as u

pm1_rect = [pm1_min, pm1_min, pm1_max, pm1_max, pm1_min] * u.mas/u.yr
pm2_rect = [pm2_min, pm2_max, pm2_max, pm2_min, pm2_min] * u.mas/u.yr
```

The following figure shows:

- Proper motion for the stars we selected along the center line of GD-1,

- The rectangle we selected, and
- The stars inside the rectangle highlighted in green.

```
import matplotlib.pyplot as plt

pm1 = centerline['pm_phi1']
pm2 = centerline['pm_phi2']
plt.plot(pm1, pm2, 'ko', markersize=0.3, alpha=0.3)

pm1 = selected['pm_phi1']
pm2 = selected['pm_phi2']
plt.plot(pm1, pm2, 'gx', markersize=0.3, alpha=0.3)

plt.plot(pm1_rect, pm2_rect, '-')

plt.xlabel('Proper motion phi1 (GD1 frame)')
plt.ylabel('Proper motion phi2 (GD1 frame)')

plt.xlim(-12, 8)
plt.ylim(-10, 10);
```



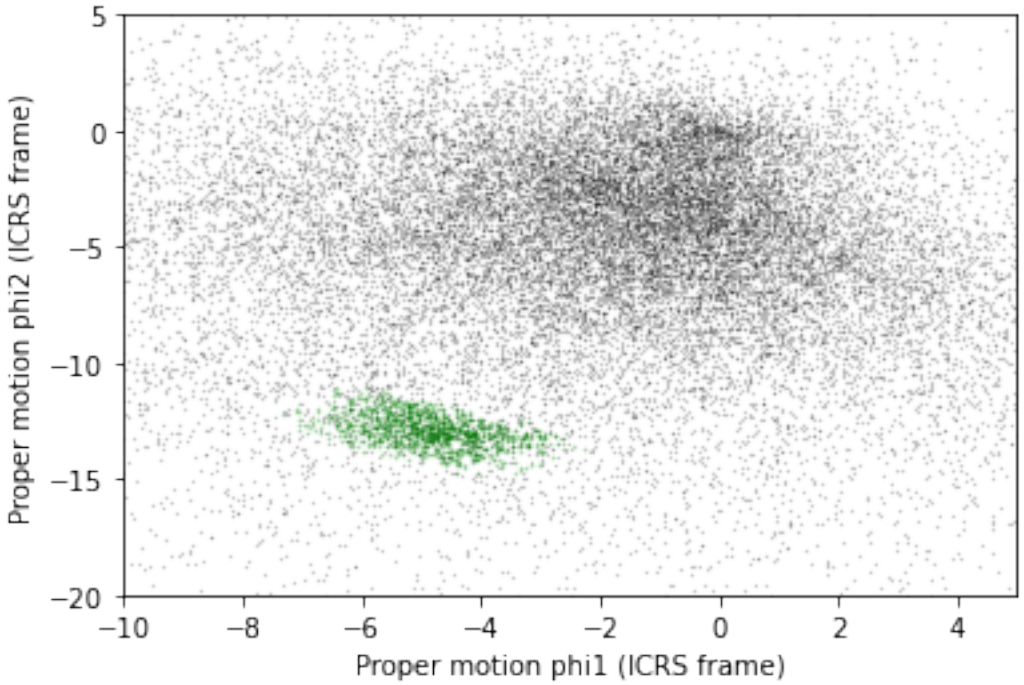
Now we'll make the same plot using proper motions in the ICRS frame, which are stored in columns `pmra` and `pmdec`.

```
pm1 = centerline['pmra']
pm2 = centerline['pmdec']
plt.plot(pm1, pm2, 'ko', markersize=0.3, alpha=0.3)

pm1 = selected['pmra']
pm2 = selected['pmdec']
plt.plot(pm1, pm2, 'gx', markersize=1, alpha=0.3)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Proper motion phi1 (ICRS frame)')
plt.ylabel('Proper motion phi2 (ICRS frame)')

plt.xlim([-10, 5])
plt.ylim([-20, 5]);
```



The proper motions of the selected stars are more spread out in this frame, which is why it was preferable to do the selection in the GD-1 frame.

But now we can define a polygon that encloses the proper motions of these stars in ICRS, and use the polygon as a selection criterion in an ADQL query.

SciPy provides a function that computes the convex hull of a set of points, which is the smallest convex polygon that contains all of the points.

To use it, I'll select columns `pmra` and `pmdec` and convert them to a NumPy array.

```
import numpy as np

points = selected[['pmra','pmdec']].to_numpy()
points.shape
```

```
(1049, 2)
```

We'll pass the points to `ConvexHull`, which returns an object that contains the results.

```
from scipy.spatial import ConvexHull

hull = ConvexHull(points)
hull
```

```
<scipy.spatial.qhull.ConvexHull at 0x7f446b1e8bb0>
```

`hull.vertices` contains the indices of the points that fall on the perimeter of the hull.

```
hull.vertices
```

```
array([ 692,  873,  141,  303,   42,  622,   45,   83,  127,  182, 1006,
        971,  967, 1001,  969,  940], dtype=int32)
```

We can use them as an index into the original array to select the corresponding rows.

```
pm_vertices = points[hull.vertices]
pm_vertices
```

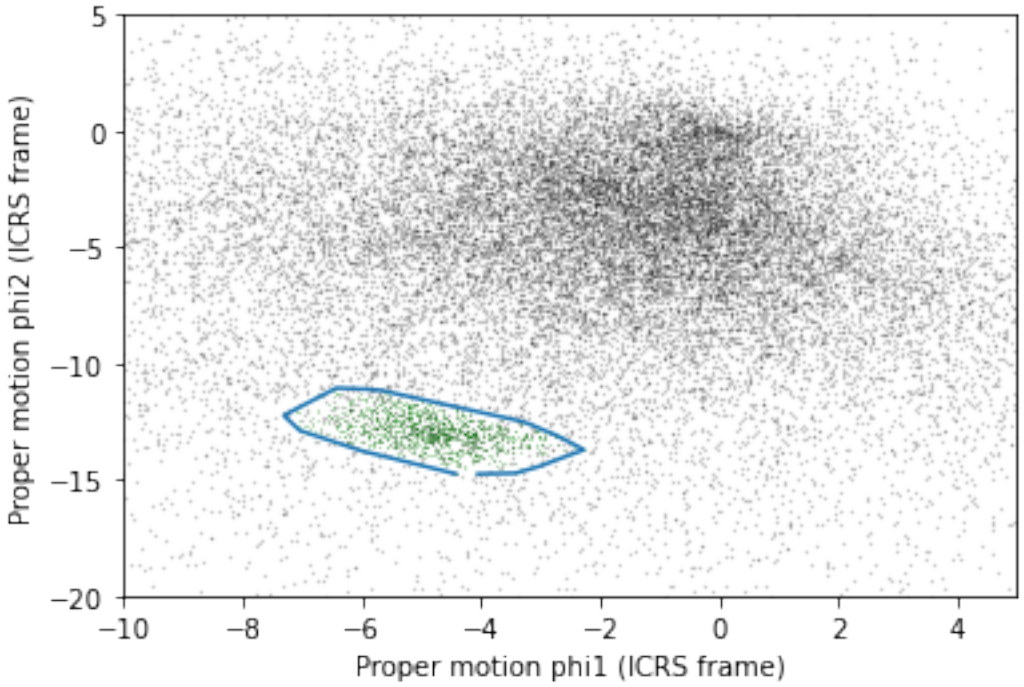
```
array([[ -4.05037121, -14.75623261],
       [ -3.41981085, -14.72365546],
       [ -3.03521988, -14.44357135],
       [ -2.26847919, -13.7140236 ],
       [ -2.61172203, -13.24797471],
       [ -2.73471401, -13.09054471],
       [ -3.19923146, -12.5942653 ],
       [ -3.34082546, -12.47611926],
       [ -5.67489413, -11.16083338],
       [ -5.95159272, -11.10547884],
       [ -6.42394023, -11.05981295],
       [ -7.09631023, -11.95187806],
       [ -7.30641519, -12.24559977],
       [ -7.04016696, -12.88580702],
       [ -6.00347705, -13.75912098],
       [ -4.42442296, -14.74641176]])
```

To plot the resulting polygon, we have to pull out the x and y coordinates.

```
pmra_poly, pmdec_poly = np.transpose(pm_vertices)
```

The following figure shows proper motion in ICRS again, along with the convex hull we just computed.

```
pm1 = centerline['pmra']
pm2 = centerline['pmdec']
plt.plot(pm1, pm2, 'ko', markersize=0.3, alpha=0.3)

pm1 = selected['pmra']
pm2 = selected['pmdec']
plt.plot(pm1, pm2, 'gx', markersize=0.3, alpha=0.3)

plt.plot(pmra_poly, pmdec_poly)

plt.xlabel('Proper motion phi1 (ICRS frame)')
plt.ylabel('Proper motion phi2 (ICRS frame)')

plt.xlim([-10, 5])
plt.ylim([-20, 5]);
```



To use `pm_vertices` as part of an ADQL query, we have to convert it to a string.

We'll use `flatten` to convert from a 2-D array to a 1-D array, and `str` to convert each element to a string.

```
t = [str(x) for x in pm_vertices.flatten()]
t
```

```
['-4.050371212154984',
 '-14.75623260987968',
 '-3.4198108491382455',
 '-14.723655456335619',
 '-3.035219883740934',
 '-14.443571352854612',
 '-2.268479190206636',
 '-13.7140235988315 54',
 '-2.611722027231764',
 '-13.247974712069263',
 '-2.7347140078529106',
 '-13.090544709622938',
 '-3.199231461993783',
 '-12.594265302440828',
 '-3.34082545787549',
 '-12.476119260818695',
 '-5.674894125178565',
 '-11.160833381392624',
 '-5.95159272432137',
 '-11.105478836426514',
 '-6.423940229776128',
 '-11.05981294804957',
 '-7.096310230579248',
 '-11.951878058650085',
 '-7.306415190921692',
 '-12.245599765990594',
```

(continues on next page)

(continued from previous page)

```
 '-7.040166963232815',
 '-12.885807024935527',
 '-6.0034770546523735',
 '-13.759120984106968',
 '-4.42442296194263',
 '-14.7464117578883']
```

Now `t` is a list of strings; we can use `join` to make a single string with commas between the elements.

```
pm_point_list = ', '.join(t)
pm_point_list
```

```
'-4.050371212154984, -14.75623260987968, -3.4198108491382455, -14.723655456335619, -3.
↪035219883740934, -14.443571352854612, -2.268479190206636, -13.714023598831554, -2.
↪611722027231764, -13.247974712069263, -2.7347140078529106, -13.090544709622938, -3.
↪199231461993783, -12.594265302440828, -3.34082545787549, -12.476119260818695, -5.
↪674894125178565, -11.160833381392624, -5.95159272432137, -11.105478836426514, -6.
↪423940229776128, -11.05981294804957, -7.096310230579248, -11.951878058650085, -7.
↪306415190921692, -12.245599765990594, -7.040166963232815, -12.885807024935527, -6.
↪0034770546523735, -13.759120984106968, -4.42442296194263, -14.7464117578883'
```

4.5 Selecting the region

Let's review how we got to this point.

1. We made an ADQL query to the Gaia server to get data for stars in the vicinity of GD-1.
2. We transformed to `GD1` coordinates so we could select stars along the centerline of GD-1.
3. We plotted the proper motion of the centerline stars to identify the bounds of the overdense region.
4. We made a mask that selects stars whose proper motion is in the overdense region.

The problem is that we downloaded data for more than 100,000 stars and selected only about 1000 of them.

It will be more efficient if we select on proper motion as part of the query. That will allow us to work with a larger region of the sky in a single query, and download less unneeded data.

This query will select on the following conditions:

- `parallax < 1`
- `bp_rp BETWEEN -0.75 AND 2`
- Coordinates within a rectangle in the GD-1 frame, transformed to ICRS.
- Proper motion with the polygon we just computed.

The first three conditions are the same as in the previous query. Only the last one is new.

Here's the rectangle in the GD-1 frame we'll select.

```
phi1_min = -70
phi1_max = -20
phi2_min = -5
phi2_max = 5
```

```
phi1_rect = [phi1_min, phi1_min, phi1_max, phi1_max] * u.deg
phi2_rect = [phi2_min, phi2_max, phi2_max, phi2_min] * u.deg
```

Here's how we transform it to ICRS, as we saw in the previous lesson.

```
import gala.coordinates as gc
import astropy.coordinates as coord

corners = gc.GD1Koposov10(phi1=phi1_rect, phi2=phi2_rect)
corners_icrs = corners.transform_to(coord.ICRS)
```

To use `corners_icrs` as part of an ADQL query, we have to convert it to a string. Here's how we do that, as we saw in the previous lesson.

```
point_base = "{point.ra.value}, {point.dec.value}"

t = [point_base.format(point=point)
     for point in corners_icrs]

point_list = ', '.join(t)
point_list
```

```
'135.30559858565638, 8.398623940157561, 126.50951508623503, 13.44494195652069, 163.
↪0173655836748, 54.24242734020255, 172.9328536286811, 46.47260492416258'
```

Now we have everything we need to assemble the query.

4.6 Assemble the query

Here's the base string we used for the query in the previous lesson.

```
query_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}))
"""
```

Exercise: Modify `query_base` by adding a new clause to select stars whose coordinates of proper motion, `pmra` and `pmdec`, fall within the polygon defined by `pm_point_list`.

```
# Solution

query_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}))
  AND 1 = CONTAINS(POINT(pmra, pmdec),
                   POLYGON({pm_point_list}))
"""
```

Here again are the columns we want to select.

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

Exercise: Use `format` to format `query_base` and define `query`, filling in the values of `columns`, `point_list`, and `pm_point_list`.

```
# Solution

query = query_base.format(columns=columns,
                          point_list=point_list,
                          pm_point_list=pm_point_list)
print(query)
```

```
SELECT
source_id, ra, dec, pmra, pmdec, parallax, parallax_error, radial_velocity
FROM gaiadr2.gaia_source
WHERE parallax < 1
  AND bp_rp BETWEEN -0.75 AND 2
  AND 1 = CONTAINS(POINT(ra, dec),
                   POLYGON(135.30559858565638, 8.398623940157561, 126.50951508623503,␣
↪13.44494195652069, 163.0173655836748, 54.24242734020255, 172.9328536286811, 46.
↪47260492416258))
  AND 1 = CONTAINS(POINT(pmra, pmdec),
                   POLYGON(-4.050371212154984, -14.75623260987968, -3.
↪4198108491382455, -14.723655456335619, -3.035219883740934, -14.443571352854612, -2.
↪268479190206636, -13.714023598831554, -2.611722027231764, -13.247974712069263, -2.
↪7347140078529106, -13.090544709622938, -3.199231461993783, -12.594265302440828, -3.
↪34082545787549, -12.476119260818695, -5.674894125178565, -11.160833381392624, -5.
↪95159272432137, -11.105478836426514, -6.423940229776128, -11.05981294804957, -7.
↪096310230579248, -11.951878058650085, -7.306415190921692, -12.245599765990594, -7.
↪040166963232815, -12.885807024935527, -6.0034770546523735, -13.759120984106968, -4.
↪42442296194263, -14.7464117578883))
```

Here's how we run it.

```
from astroquery.gaia import Gaia

job = Gaia.launch_job_async(query)
print(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
INFO: Query finished. [astroquery.utils.tap.core]
<Table length=7346>
      name       dtype    unit                                  description
↪              n_bad
--------------- ------- -------- ------------------------------------------------------
↪-------------- -----
```

(continues on next page)

(continued from previous page)

```
      source_id   int64          Unique source identifier (unique within a particular␣
↪Data Release)      0
             ra float64      deg                                                      ␣
↪Right ascension     0
            dec float64      deg                                                      ␣
↪  Declination     0
           pmra float64 mas / yr                          Proper motion in right␣
↪ascension direction     0
          pmdec float64 mas / yr                              Proper motion in␣
↪declination direction     0
       parallax float64      mas                                                      ␣
↪      Parallax     0
 parallax_error float64      mas                                           Standard␣
↪error of parallax     0
radial_velocity float64   km / s                                                      ␣
↪Radial velocity  7295
Jobid: 16031327462370
Phase: COMPLETED
Owner: None
Output file: async_20201019143906.vot
Results: None
```

And get the results.

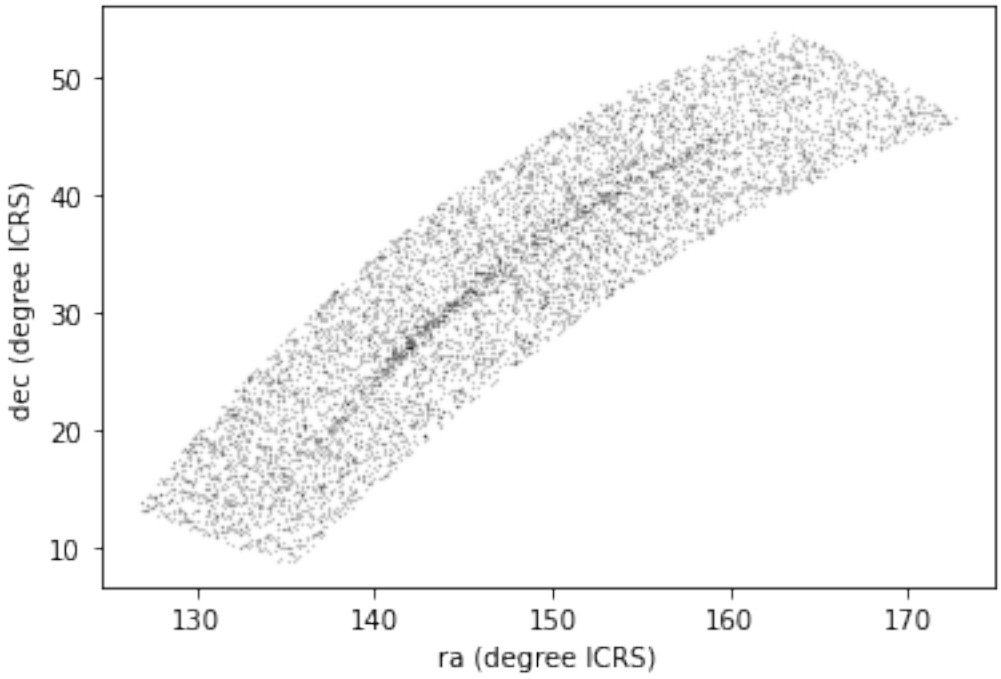
```
candidate_table = job.get_results()
len(candidate_table)
```

```
7346
```

4.7 Plotting one more time

Let's see what the results look like.

```
x = candidate_table['ra']
y = candidate_table['dec']
plt.plot(x, y, 'ko', markersize=0.3, alpha=0.3)

plt.xlabel('ra (degree ICRS)')
plt.ylabel('dec (degree ICRS)');
```



Here we can see why it was useful to transform these coordinates. In ICRS, it is more difficult to identity the stars near the centerline of GD-1.

So, before we move on to the next step, let's collect the code we used to transform the coordinates and make a Pandas `DataFrame`:

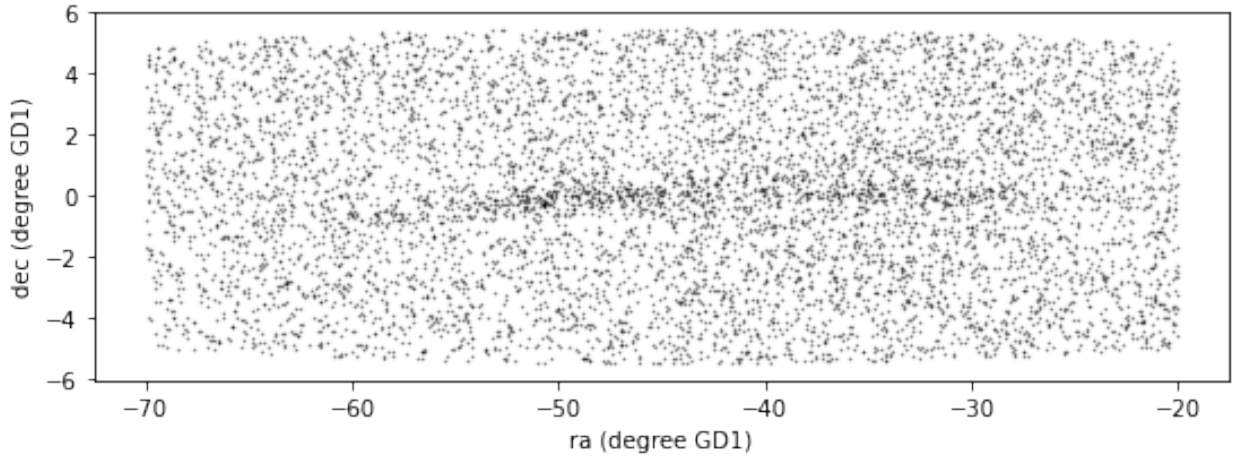
```
from pyia import GaiaData

def make_dataframe(table):
    """Transform coordinates from ICRS to GD-1 frame.

    table: Astropy Table

    returns: Pandas DataFrame
    """
    gaia_data = GaiaData(table)

    c_sky = gaia_data.get_skycoord(distance=8*u.kpc,
                                   radial_velocity=0*u.km/u.s)
    c_gd1 = gc.reflex_correct(
                c_sky.transform_to(gc.GD1Koposov10))

    df = table.to_pandas()
    df['phi1'] = c_gd1.phi1
    df['phi2'] = c_gd1.phi2
    df['pm_phi1'] = c_gd1.pm_phi1_cosphi2
    df['pm_phi2'] = c_gd1.pm_phi2
    return df
```

Here's how we can use this function:

```
candidate_df = make_dataframe(candidate_table)
```

And let's see the results.

```
x = candidate_df['phi1']
y = candidate_df['phi2']

plt.plot(x, y, 'ko', markersize=0.5, alpha=0.5)

plt.xlabel('ra (degree GD1)')
plt.ylabel('dec (degree GD1)');
```



We're starting to see GD-1 more clearly.

We can compare this figure with one of these panels in Figure 1 from the original paper:

The top panel shows stars selected based on proper motion only, so it is comparable to our figure (although notice that it covers a wider region).

In the next lesson, we will use photometry data from Pan-STARRS to do a second round of filtering, and see if we can replicate the bottom panel.

We'll also learn how to add annotations like the ones in the figure from the paper, and customize the style of the figure to present the results clearly and compellingly.

4.8 Saving the DataFrame

Let's save this `DataFrame` so we can pick up where we left off without running this query again.

```
!rm -f gd1_candidates.hdf5
```

```
filename = 'gd1_candidates.hdf5'

candidate_df.to_hdf(filename, 'candidate_df')
```

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

```
!ls -lh gd1_candidates.hdf5
```

```
-rw-rw-r-- 1 downey downey 756K Oct 19 14:39 gd1_candidates.hdf5
```

If you are using Windows, `ls` might not work; in that case, try:

```
!dir gd1_candidates.hdf5
```

4.9 CSV

Pandas can write a variety of other formats, which you can read about here.

We won't cover all of them, but one other important one is CSV, which stands for "comma-separated values".

CSV is a plain-text format with minimal formatting requirements, so it can be read and written by pretty much any tool that works with data. In that sense, it is the "least common denominator" of data formats.

However, it has an important limitation: some information about the data gets lost in translation, notably the data types. If you read a CSV file from someone else, you might need some additional information to make sure you are getting it right.

Also, CSV files tend to be big, and slow to read and write.

With those caveats, here's how to write one:

```
candidate_df.to_csv('gd1_candidates.csv')
```

We can check the file size like this:

```
!ls -lh gd1_candidates.csv
```

```
-rw-rw-r-- 1 downey downey 1.6M Oct 19 14:39 gd1_candidates.csv
```

The CSV file about 2 times bigger than the HDF5 file (so that's not that bad, really).

We can see the first few lines like this:

```
!head -3 gd1_candidates.csv
```

```
,source_id,ra,dec,pmra,pmdec,parallax,parallax_error,radial_velocity,phi1,phi2,pm_
↪phi1,pm_phi2
0,635559124339440000,137.58671691646745,19.1965441084838,-3.770521900009566,-12.
↪490481778113859,0.7913934419894347,0.2717538145759051,,-59.63048941944396,-1.
↪21648525150429,-7.361362712556612,-0.5926328820420083
1,635860218726658176,138.5187065217173,19.09233926905897,-5.941679495793577,-11.
↪346409129876392,0.30745551377348623,0.19946557779138105,,-59.247329893833296,-2.
↪0160784008206476,-7.527126084599517,1.7487794924398758
```

The CSV file contains the names of the columns, but not the data types.

We can read the CSV file back like this:

```
read_back_csv = pd.read_csv('gd1_candidates.csv')
```

Let's compare the first few rows of `candidate_df` and `read_back_csv`

```
candidate_df.head(3)
```

```
            source_id          ra        dec      pmra      pmdec  parallax  \
0  635559124339440000  137.586717  19.196544 -3.770522 -12.490482  0.791393
1  635860218726658176  138.518707  19.092339 -5.941679 -11.346409  0.307456
```

(continues on next page)

(continued from previous page)

```
2  635674126383965568  138.842874  19.031798 -3.897001 -12.702780  0.779463

   parallax_error  radial_velocity       phi1      phi2   pm_phi1   pm_phi2
0        0.271754              NaN -59.630489 -1.216485 -7.361363 -0.592633
1        0.199466              NaN -59.247330 -2.016078 -7.527126  1.748779
2        0.223692              NaN -59.133391 -2.306901 -7.560608 -0.741800
```

```
read_back_csv.head(3)
```

```
   Unnamed: 0           source_id          ra        dec      pmra      pmdec  \
0           0  635559124339440000  137.586717  19.196544 -3.770522 -12.490482
1           1  635860218726658176  138.518707  19.092339 -5.941679 -11.346409
2           2  635674126383965568  138.842874  19.031798 -3.897001 -12.702780

   parallax  parallax_error  radial_velocity       phi1      phi2   pm_phi1  \
0  0.791393        0.271754              NaN -59.630489 -1.216485 -7.361363
1  0.307456        0.199466              NaN -59.247330 -2.016078 -7.527126
2  0.779463        0.223692              NaN -59.133391 -2.306901 -7.560608

    pm_phi2
0 -0.592633
1  1.748779
2 -0.741800
```

Notice that the index in `candidate_df` has become an unnamed column in `read_back_csv`. The Pandas functions for writing and reading CSV files provide options to avoid that problem, but this is an example of the kind of thing that can go wrong with CSV files.

4.10 Summary

In the previous lesson we downloaded data for a large number of stars and then selected a small fraction of them based on proper motion.

In this lesson, we improved this process by writing a more complex query that uses the database to select stars based on proper motion. This process requires more computation on the Gaia server, but then we're able to either:

1. Search the same region and download less data, or
2. Search a larger region while still downloading a manageable amount of data.

In the next lesson, we'll learn about the databased `JOIN` operation and use it to download photometry data from Pan-STARRS.

4.11 Best practices

- When possible, "move the computation to the data"; that is, do as much of the work as possible on the database server before downloading the data.
- For most applications, saving data in FITS or HDF5 is better than CSV. FITS and HDF5 are binary formats, so the files are usually smaller, and they store metadata, so you don't lose anything when you read the file back.
- On the other hand, CSV is a "least common denominator" format; that is, it can be read by practically any application that works with data.

CHAPTER

FIVE

CHAPTER 5

This is the fifth in a series of notebooks related to astronomy data.

As a continuing example, we will replicate part of the analysis in a recent paper, "Off the beaten path: Gaia reveals GD-1 stars outside of the main stream" by Adrian M. Price-Whelan and Ana Bonaca.

Picking up where we left off, the next step in the analysis is to select candidate stars based on photometry. The following figure from the paper is a color-magnitude diagram for the stars selected based on proper motion:

In red is a theoretical isochrone, showing where we expect the stars in GD-1 to fall based on the metallicity and age of their original globular cluster.

By selecting stars in the shaded area, we can further distinguish the main sequence of GD-1 from younger background stars.

5.1 Outline

Here are the steps in this notebook:

1. We'll reload the candidate stars we identified in the previous notebook.
2. Then we'll run a query on the Gaia server that uploads the table of candidates and uses a `JOIN` operation to select photometry data for the candidate stars.
3. We'll write the results to a file for use in the next notebook.

After completing this lesson, you should be able to

- Upload a table to the Gaia server.
- Write ADQL queries involving `JOIN` operations.

5.2 Installing libraries

If you are running this notebook on Colab, you can run the following cell to install Astroquery and a the other libraries we'll use.

If you are running this notebook on your own computer, you might have to install these libraries yourself.

If you are using this notebook as part of a Carpentries workshop, you should have received setup instructions.

TODO: Add a link to the instructions.

```
# If we're running on Colab, install libraries

import sys
IN_COLAB = 'google.colab' in sys.modules

if IN_COLAB:
    !pip install astroquery astro-gala pyia python-wget
```

5.3 Reloading the data

The following cell downloads the data from the previous notebook.

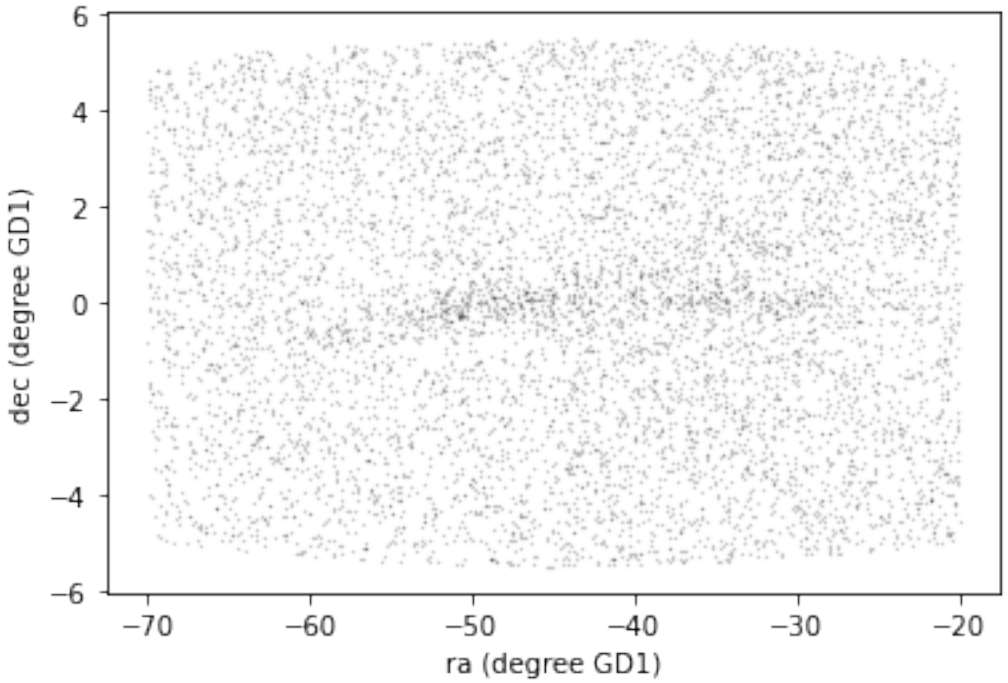
```
import os
from wget import download

filename = 'gd1_candidates.hdf5'
path = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(path+filename))
```

And we can read it back.

```
import pandas as pd

candidate_df = pd.read_hdf(filename, 'candidate_df')
```

`candidate_df` is the Pandas DataFrame that contains results from the query in the previous notebook, which selects stars likely to be in GD-1 based on proper motion. It also includes position and proper motion transformed to the ICRS frame.

```
import matplotlib.pyplot as plt

x = candidate_df['phi1']
y = candidate_df['phi2']

plt.plot(x, y, 'ko', markersize=0.3, alpha=0.3)

plt.xlabel('ra (degree GD1)')
plt.ylabel('dec (degree GD1)');
```



This is the same figure we saw at the end of the previous notebook. GD-1 is visible against the background stars, but we will be able to see it more clearly after selecting based on photometry data.

5.4 Getting photometry data

The Gaia dataset contains some photometry data, including the variable `bp_rp`, which we used in the original query to select stars with BP - RP color between -0.75 and 2.

Selecting stars with `bp-rp` less than 2 excludes many class M dwarf stars, which are low temperature, low luminosity. A star like that at GD-1's distance would be hard to detect, so if it is detected, it it more likely to be in the foreground.

Now, to select stars with the age and metal richness we expect in GD-1, we will use `g - i` color and apparent `g`-band magnitude, which are available from the Pan-STARRS survey.

Conveniently, the Gaia server provides data from Pan-STARRS as a table in the same database we have been using, so we can access it by making ADQL queries.

In general, looking up a star from the Gaia catalog and finding the corresponding star in the Pan-STARRS catalog is not easy. This kind of cross matching is not always possible, because a star might appear in one catalog and not the other. And even when both stars are present, there might not be a clear one-to-one relationship between stars in the two catalogs.

Fortunately, smart people have worked on this problem, and the Gaia database includes cross-matching tables that suggest a best neighbor in the Pan-STARRS catalog for many stars in the Gaia catalog.

This document describes the cross matching process. Briefly, it uses a cone search to find possible matches in approximately the right position, then uses attributes like color and magnitude to choose pairs of stars most likely to be identical.

So the hard part of cross-matching has been done for us. However, using the results is a little tricky.

But, it is also an opportunity to learn about one of the most important tools for working with databases: "joining" tables.

In general, a "join" is an operation where you match up records from one table with records from another table using as a "key" a piece of information that is common to both tables, usually some kind of ID code.

In this example:

- Stars in the Gaia dataset are identified by `source_id`.
- Stars in the Pan-STARRS dataset are identified by `obj_id`.

For each candidate star we have selected so far, we have the `source_id`; the goal is to find the `obj_id` for the same star (we hope) in the Pan-STARRS catalog.

To do that we will:

1. Make a table that contains the `source_id` for each candidate star and upload the table to the Gaia server;
2. Use the `JOIN` operator to look up each `source_id` in the `gaiadr2.panstarrs1_best_neighbour` table, which contains the `obj_id` of the best match for each star in the Gaia catalog; then
3. Use the `JOIN` operator again to look up each `obj_id` in the `panstarrs1_original_valid` table, which contains the Pan-STARRS photometry data we want.

Let's start with the first step, uploading a table.

5.5 Preparing a table for uploading

For each candidate star, we want to find the corresponding row in the `gaiadr2.panstarrs1_best_neighbour` table.

In order to do that, we have to:

1. Write the table in a local file as an XML VOTable, which is a format suitable for transmitting a table over a network.
2. Write an ADQL query that refers to the uploaded table.
3. Change the way we submit the job so it uploads the table before running the query.

The first step is not too difficult because Astropy provides a function called `writeto` that can write a `Table` in `XML`.

The documentation of this process is here.

First we have to convert our Pandas `DataFrame` to an Astropy `Table`.

```
from astropy.table import Table

candidate_table = Table.from_pandas(candidate_df)
type(candidate_table)
```

```
astropy.table.table.Table
```

To write the file, we can use `Table.write` with `format='votable'`, as described here.

```
table = candidate_table[['source_id']]
table.write('candidate_df.xml', format='votable', overwrite=True)
```

Notice that we select a single column from the table, `source_id`. We could write the entire table to a file, but that would take longer to transmit over the network, and we really only need one column.

This process, taking a structure like a `Table` and translating it into a form that can be transmitted over a network, is called serialization.

XML is one of the most common serialization formats. One nice feature is that XML data is plain text, as opposed to binary digits, so you can read the file we just wrote:

```
!head candidate_df.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Produced with astropy.io.votable version 4.0.1.post1
     http://www.astropy.org/ -->
<VOTABLE version="1.4" xmlns="http://www.ivoa.net/xml/VOTable/v1.4" xmlns:xsi="http://
↪www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://www.ivoa.
↪net/xml/VOTable/v1.4">
 <RESOURCE type="results">
  <TABLE>
   <FIELD ID="source_id" datatype="long" name="source_id"/>
   <DATA>
    <TABLEDATA>
     <TR>
```

XML is a general format, so different XML files contain different kinds of data. In order to read an XML file, it's not enough to know that it's XML; you also have to know the data format, which is called a schema.

In this example, the schema is VOTable; notice that one of the first tags in the file specifies the schema, and even includes the URL where you can get its definition.

So this is an example of a self-documenting format.

A drawback of XML is that it tends to be big, which is why we wrote just the `source_id` column rather than the whole table. The size of the file is about 750 KB, so that's not too bad.

```
!ls -lh candidate_df.xml
```

```
-rw-rw-r-- 1 downey downey 396K Oct 19 14:48 candidate_df.xml
```

If you are using Windows, `ls` might not work; in that case, try:

```
!dir candidate_df.xml
```

Exercise: There's a gotcha here we want to warn you about. Why do you think we used double brackets to specify the column we wanted? What happens if you use single brackets?

Run these cells to find out.

```
table = candidate_table[['source_id']]
type(table)
```

```
astropy.table.table.Table
```

```
column = candidate_table['source_id']
type(column)
```

```
astropy.table.column.Column
```

```
# writeto(column, 'candidate_df.xml')
```

5.6 Uploading a table

The next step is to upload this table to the Gaia server and use it as part of a query.

Here's the documentation that explains how to run a query with an uploaded table.

In the spirit of incremental development and testing, let's start with the simplest possible query.

```
query = """SELECT *
FROM tap_upload.candidate_df
"""
```

This query downloads all rows and all columns from the uploaded table. The name of the table has two parts: `tap_upload` specifies a table that was uploaded using TAP+ (remember that's the name of the protocol we're using to talk to the Gaia server).

And `candidate_df` is the name of the table, which we get to choose (unlike `tap_upload`, which we didn't get to choose).

Here's how we run the query:

```
from astroquery.gaia import Gaia

job = Gaia.launch_job_async(query=query,
                            upload_resource='candidate_df.xml',
                            upload_table_name='candidate_df')
```

```
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
INFO: Query finished. [astroquery.utils.tap.core]
```

`upload_resource` specifies the name of the file we want to upload, which is the file we just wrote.

`upload_table_name` is the name we assign to this table, which is the name we used in the query.

And here are the results:

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

```
<Table length=7346>
    source_id
      int64
------------------
635559124339440000
635860218726658176
635674126383965568
635535454774983040
635497276810313600
635614168640132864
```

(continues on next page)

(continued from previous page)

```
635821843194387840
635551706931167104
635518889086133376
635580294233854464
                   ...
612282738058264960
612485911486166656
612386332668697600
612296172717818624
612250375480101760
612394926899159168
612288854091187712
612428870024913152
612256418500423168
612429144902815104
```

If things go according to plan, the result should contain the same rows and columns as the uploaded table.

```
len(candidate_table), len(results)
```

```
(7346, 7346)
```

```
set(candidate_table['source_id']) == set(results['source_id'])
```

```
True
```

In this example, we uploaded a table and then downloaded it again, so that's not too useful.

But now that we can upload a table, we can join it with other tables on the Gaia server.

5.7 Joining with an uploaded table

Here's the first example of a query that contains a `JOIN` clause.

```
query1 = """SELECT *
FROM gaiadr2.panstarrs1_best_neighbour as best
JOIN tap_upload.candidate_df as candidate_df
ON best.source_id = candidate_df.source_id
"""
```

Let's break that down one clause at a time:

- `SELECT *` means we will download all columns from both tables.
- `FROM gaiadr2.panstarrs1_best_neighbour as best` means that we'll get the columns from the Pan-STARRS best neighbor table, which we'll refer to using the short name `best`.
- `JOIN tap_upload.candidate_df as candidate_df` means that we'll also get columns from the uploaded table, which we'll refer to using the short name `candidate_df`.
- `ON best.source_id = candidate_df.source_id` specifies that we will use `source_id` to match up the rows from the two tables.

Here's the documentation of the best neighbor table.

Let's run the query:

```
job1 = Gaia.launch_job_async(query=query1,
                        upload_resource='candidate_df.xml',
                        upload_table_name='candidate_df')
```

```
INFO: Query finished. [astroquery.utils.tap.core]
```

And get the results.

```
results1 = job1.get_results()
results1
```

```
<Table length=3724>
    source_id      original_ext_source_id ...    source_id_2
                                          ...
      int64                 int64         ...       int64
------------------ --------------------- ... ------------------
635860218726658176     130911385187671349 ... 635860218726658176
635674126383965568     130831388428488720 ... 635674126383965568
635535454774983040     130631378377657369 ... 635535454774983040
635497276810313600     130811380445631930 ... 635497276810313600
635614168640132864     130571395922140135 ... 635614168640132864
635598607974369792     130341392091279513 ... 635598607974369792
635737661835496576     131001399333502136 ... 635737661835496576
635850945892748672     132011398654934147 ... 635850945892748672
635600532119713664     130421392285893623 ... 635600532119713664
                ...                    ... ...                ...
612241781249124608     129751343755995561 ... 612241781249124608
612332147361443072     130141341458538777 ... 612332147361443072
612426744016802432     130521346852465656 ... 612426744016802432
612331739340341760     130111341217793839 ... 612331739340341760
612282738058264960     129741340445933519 ... 612282738058264960
612386332668697600     130351354570219774 ... 612386332668697600
612296172717818624     129691338006168780 ... 612296172717818624
612250375480101760     129741346475897464 ... 612250375480101760
612394926899159168     130581355199751795 ... 612394926899159168
612256418500423168     129931349075297310 ... 612256418500423168
```

This table contains all of the columns from the best neighbor table, plus the single column from the uploaded table.

```
results1.colnames
```

```
['source_id',
 'original_ext_source_id',
 'angular_distance',
 'number_of_neighbours',
 'number_of_mates',
 'best_neighbour_multiplicity',
 'gaia_astrometric_params',
 'source_id_2']
```

Because one of the column names appears in both tables, the second instance of `source_id` has been appended with the suffix `_2`.

The length of the results table is about 2000, which means we were not able to find matches for all stars in the list of candidate_df.

```
len(results1)
```

```
3724
```

To get more information about the matching process, we can inspect `best_neighbour_multiplicity`, which indicates for each star in Gaia how many stars in Pan-STARRS are equally likely matches.

For this kind of data exploration, we'll convert a column from the table to a Pandas `Series` so we can use `value_counts`, which counts the number of times each value appears in a `Series`, like a histogram.

```
import pandas as pd

nn = pd.Series(results1['best_neighbour_multiplicity'])
nn.value_counts()
```

```
1    3724
dtype: int64
```

The result shows that `1` is the only value in the `Series`, appearing xxx times.

That means that in every case where a match was found, the matching algorithm identified a single neighbor as the most likely match.

Similarly, `number_of_mates` indicates the number of other stars in Gaia that match with the same star in Pan-STARRS.

```
nm = pd.Series(results1['number_of_mates'])
nm.value_counts()
```

```
0    3724
dtype: int64
```

For this set of candidate_df, almost all of the stars we've selected from Pan-STARRS are only matched with a single star in the Gaia catalog.

Detail The table also contains `number_of_neighbors` which is the number of stars in Pan-STARRS that match in terms of position, before using other critieria to choose the most likely match.

5.8 Getting the photometry data

The most important column in `results1` is `original_ext_source_id` which is the `obj_id` we will use to look up the likely matches in Pan-STARRS to get photometry data.

The process is similar to what we just did to look up the matches. We will:

1. Make a table that contains `source_id` and `original_ext_source_id`.
2. Write the table to an XML VOTable file.
3. Write a query that joins the uploaded table with `gaiadr2.panstarrs1_original_valid` and selects the photometry data we want.
4. Run the query using the uploaded table.

Since we've done everything here before, we'll do these steps as an exercise.

Exercise: Select `source_id` and `original_ext_source_id` from `results1` and write the resulting table as a file named `external.xml`.

```
# Solution

table = results1[['source_id', 'original_ext_source_id']]
table.write('external.xml', format='votable', overwrite=True)
```

Use `!head` to confirm that the file exists and contains an XML VOTable.

```
!head external.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Produced with astropy.io.votable version 4.0.1.post1
     http://www.astropy.org/ -->
<VOTABLE version="1.4" xmlns="http://www.ivoa.net/xml/VOTable/v1.4" xmlns:xsi="http://
↪www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://www.ivoa.
↪net/xml/VOTable/v1.4">
 <RESOURCE type="results">
  <TABLE>
   <FIELD ID="source_id" datatype="long" name="source_id" ucd="meta.id;meta.main">
    <DESCRIPTION>
     Unique Gaia source identifier
    </DESCRIPTION>
```

Exercise: Read the documentation of the Pan-STARRS table and make note of `obj_id`, which contains the object IDs we'll use to find the rows we want.

Write a query that uses each value of `original_ext_source_id` from the uploaded table to find a row in `gaiadr2.panstarrs1_original_valid` with the same value in `obj_id`, and select all columns from both tables.

Suggestion: Develop and test your query incrementally. For example:

1. Write a query that downloads all columns from the uploaded table. Test to make sure we can read the uploaded table.
2. Write a query that downloads the first 10 rows from `gaiadr2.panstarrs1_original_valid`. Test to make sure we can access Pan-STARRS data.
3. Write a query that joins the two tables and selects all columns. Test that the join works as expected.

As a bonus exercise, write a query that joins the two tables and selects just the columns we need:

- `source_id` from the uploaded table
- `g_mean_psf_mag` from `gaiadr2.panstarrs1_original_valid`
- `i_mean_psf_mag` from `gaiadr2.panstarrs1_original_valid`

Hint: When you select a column from a join, you have to specify which table the column is in.

```
# Solution

query2 = """SELECT *
FROM tap_upload.external as external
"""
```

```
# Solution

query2 = """SELECT TOP 10 *
FROM gaiadr2.panstarrs1_original_valid
"""
```

```
# Solution

query2 = """SELECT *
FROM gaiadr2.panstarrs1_original_valid as ps
JOIN tap_upload.external as external
ON ps.obj_id = external.original_ext_source_id
"""
```

```
# Solution

query2 = """SELECT
external.source_id, ps.g_mean_psf_mag, ps.i_mean_psf_mag
FROM gaiadr2.panstarrs1_original_valid as ps
JOIN tap_upload.external as external
ON ps.obj_id = external.original_ext_source_id
"""
```

```
print(query2)
```

```
SELECT
external.source_id, ps.g_mean_psf_mag, ps.i_mean_psf_mag
FROM gaiadr2.panstarrs1_original_valid as ps
JOIN tap_upload.external as external
ON ps.obj_id = external.original_ext_source_id
```

```
job2 = Gaia.launch_job_async(query=query2,
                        upload_resource='external.xml',
                        upload_table_name='external')
```

```
INFO: Query finished. [astroquery.utils.tap.core]
```

```
results2 = job2.get_results()
results2
```

```
<Table length=3724>
    source_id       g_mean_psf_mag   i_mean_psf_mag
                                           mag
      int64            float64          float64
------------------ ----------------- -----------------
635860218726658176 17.8978004455566 17.5174007415771
635674126383965568 19.2873001098633 17.6781005859375
635535454774983040 16.9237995147705  16.478099822998
635497276810313600 19.9242000579834 18.3339996337891
635614168640132864 16.1515998840332 14.6662998199463
635598607974369792 16.5223999023438 16.1375007629395
635737661835496576 14.5032997131348 13.9849004745483
635850945892748672 16.5174999237061 16.0450000762939
635600532119713664 20.4505996704102 19.5177001953125
                ...               ...              ...
612241781249124608 20.2343997955322 18.6518001556396
612332147361443072 21.3848991394043 20.3076000213623
612426744016802432 17.8281002044678 17.4281005859375
612331739340341760 21.8656997680664 19.5223007202148
612282738058264960 22.5151996612549 19.9743995666504
612386332668697600 19.3792991638184 17.9923000335693
```

(continues on next page)

(continued from previous page)

```
612296172717818624 17.4944000244141  16.926700592041
612250375480101760 15.3330001831055 14.6280002593994
612394926899159168 16.4414005279541 15.8212003707886
612256418500423168 20.8715991973877 19.9612007141113
```

Challenge exercise

Do both joins in one query.

There's an example here you could start with.

5.9 Write the data

Since we have the data in an Astropy `Table`, let's store it in a FITS file.

```
filename = 'gd1_photo.fits'
results2.write(filename, overwrite=True)
```

We can check that the file exists, and see how big it is.

```
!ls -lh gd1_photo.fits
```

```
-rw-rw-r-- 1 downey downey 96K Oct 19 14:49 gd1_photo.fits
```

At around 175 KB, it is smaller than some of the other files we've been working with.

If you are using Windows, `ls` might not work; in that case, try:

```
!dir gd1_photo.fits
```

5.10 Summary

In this notebook, we used database `JOIN` operations to select photometry data for the stars we've identified as candidates to be in GD-1.

In the next notebook, we'll use this data for a second round of selection, identifying stars that have photometry data consistent with GD-1.

5.11 Best practice

- Use `JOIN` operations to combine data from multiple tables in a databased, using some kind of identifier to match up records from one table with records from another.
- This is another example of a practice we saw in the previous notebook, moving the computation to the data.

CHAPTER

SIX

CHAPTER 6

This is the sixth in a series of notebooks related to astronomy data.

As a continuing example, we will replicate part of the analysis in a recent paper, "Off the beaten path: Gaia reveals GD-1 stars outside of the main stream" by Adrian M. Price-Whelan and Ana Bonaca.

In the previous lesson we downloaded photometry data from Pan-STARRS, which is available from the same server we've been using to get Gaia data.

The next step in the analysis is to select candidate stars based on the photometry data. The following figure from the paper is a color-magnitude diagram for the stars selected based on proper motion:

In red is a theoretical isochrone, showing where we expect the stars in GD-1 to fall based on the metallicity and age of their original globular cluster.

By selecting stars in the shaded area, we can further distinguish the main sequence of GD-1 from younger background stars.

6.1 Outline

Here are the steps in this notebook:

1. We'll reload the data from the previous notebook and make a color-magnitude diagram.
2. Then we'll specify a polygon in the diagram that contains stars with the photometry we expect.
3. Then we'll merge the photometry data with the list of candidate stars, storing the result in a Pandas `DataFrame`.

After completing this lesson, you should be able to

- Use Matplotlib to specify a `Polygon` and determine which points fall inside it.
- Use Pandas to merge data from multiple `DataFrames`, much like a database `JOIN` operation.

6.2 Installing libraries

If you are running this notebook on Colab, you can run the following cell to install Astroquery and a the other libraries we'll use.

If you are running this notebook on your own computer, you might have to install these libraries yourself.

If you are using this notebook as part of a Carpentries workshop, you should have received setup instructions.

TODO: Add a link to the instructions.

```
# If we're running on Colab, install libraries

import sys
IN_COLAB = 'google.colab' in sys.modules

if IN_COLAB:
    !pip install astroquery astro-gala pyia python-wget
```

6.3 Reload the data

The following cell downloads the photometry data we created in the previous notebook.

```
import os
from wget import download

filename = 'gd1_photo.fits'
filepath = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(filepath+filename))
```

Now we can read the data back into an Astropy `Table`.

```
from astropy.table import Table

photo_table = Table.read(filename)
```

6.4 Plotting photometry data

Now that we have photometry data from Pan-STARRS, we can replicate the color-magnitude diagram from the original paper:

The y-axis shows the apparent magnitude of each source with the g filter.

The x-axis shows the difference in apparent magnitude between the g and i filters, which indicates color.

Stars with lower values of (g-i) are brighter in g-band than in i-band, compared to other stars, which means they are bluer.

Stars in the lower-left quadrant of this diagram are less bright and less metallic than the others, which means they are likely to be older.

Since we expect the stars in GD-1 to be older than the background stars, the stars in the lower-left are more likely to be in GD-1.

```
import matplotlib.pyplot as plt

def plot_cmd(table):
    """Plot a color magnitude diagram.

    table: Table or DataFrame with photometry data
    """
    y = table['g_mean_psf_mag']
    x = table['g_mean_psf_mag'] - table['i_mean_psf_mag']
```

(continues on next page)

(continued from previous page)

```
    plt.plot(x, y, 'ko', markersize=0.3, alpha=0.3)

    plt.xlim([0, 1.5])
    plt.ylim([14, 22])
    plt.gca().invert_yaxis()

    plt.ylabel('$g_0$')
    plt.xlabel('$(g-i)_0$')
```

`plot_cmd` uses a new function, `invert_yaxis`, to invert the `y` axis, which is conventional when plotting magnitudes, since lower magnitude indicates higher brightness.

`invert_yaxis` is a little different from the other functions we've used. You can't call it like this:

```
plt.invert_yaxis()          # doesn't work
```

You have to call it like this:

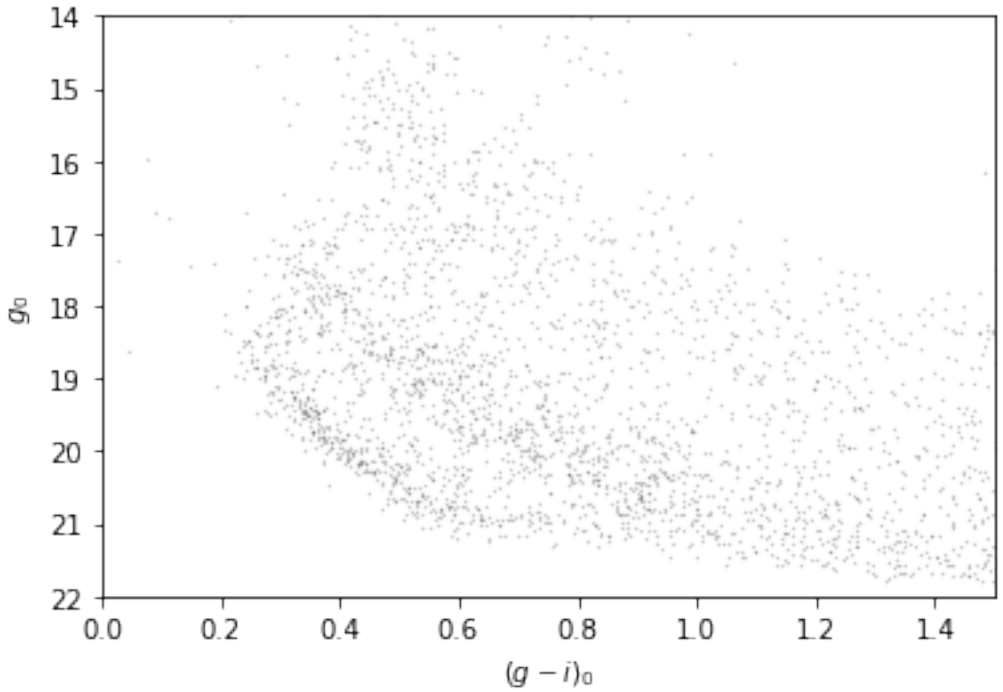
```
plt.gca().invert_yaxis()          # works
```

`gca` stands for "get current axis". It returns an object that represents the axes of the current figure, and that object provides `invert_yaxis`.

In case anyone asks: The most likely reason for this inconsistency in the interface is that `invert_yaxis` is a lesser-used function, so it's not made available at the top level of the interface.

Here's what the results look like.

```
plot_cmd(photo_table)
```



Our figure does not look exactly like the one in the paper because we are working with a smaller region of the sky, so we don't have as many stars. But we can see an overdense region in the lower left that contains stars with the

photometry we expect for GD-1.

The authors of the original paper derive a detailed polygon that defines a boundary between stars that are likely to be in GD-1 or not.

As a simplification, we'll choose a boundary by eye that seems to contain the overdense region.

6.5 Drawing a polygon

Matplotlib provides a function called `ginput` that lets us click on the figure and make a list of coordinates.

It's a little tricky to use `ginput` in a Jupyter notebook.Before calling `plt.ginput` we have to tell Matplotlib to use `TkAgg` to draw the figure in a new window.

When you run the following cell, a figure should appear in a new window. Click on it 10 times to draw a polygon around the overdense area. A red cross should appear where you click.

```
import matplotlib as mpl

if IN_COLAB:
    coords = None
else:
    mpl.use('TkAgg')
    plot_cmd(photo_table)
    coords = plt.ginput(10)
    mpl.use('agg')
```

The argument to `ginput` is the number of times the user has to click on the figure.

The result from `ginput` is a list of coordinate pairs.

```
coords
```

```
[(0.2150537634408602, 17.548197203826344),
 (0.3897849462365591, 18.94628403237675),
 (0.5376344086021505, 19.902869757174393),
 (0.7034050179211468, 20.601913171449596),
 (0.8288530465949819, 21.300956585724798),
 (0.6630824372759856, 21.52170713760118),
 (0.4301075268817204, 20.785871964679913),
 (0.27329749103942647, 19.71891096394408),
 (0.17473118279569888, 18.688741721854306),
 (0.17473118279569888, 17.95290654893304)]
```

If `ginput` doesn't work for you, you could use the following coordinates.

```
if coords is None:
    coords = [(0.2, 17.5),
              (0.2, 19.5),
              (0.65, 22),
              (0.75, 21),
              (0.4, 19),
              (0.4, 17.5)]
```

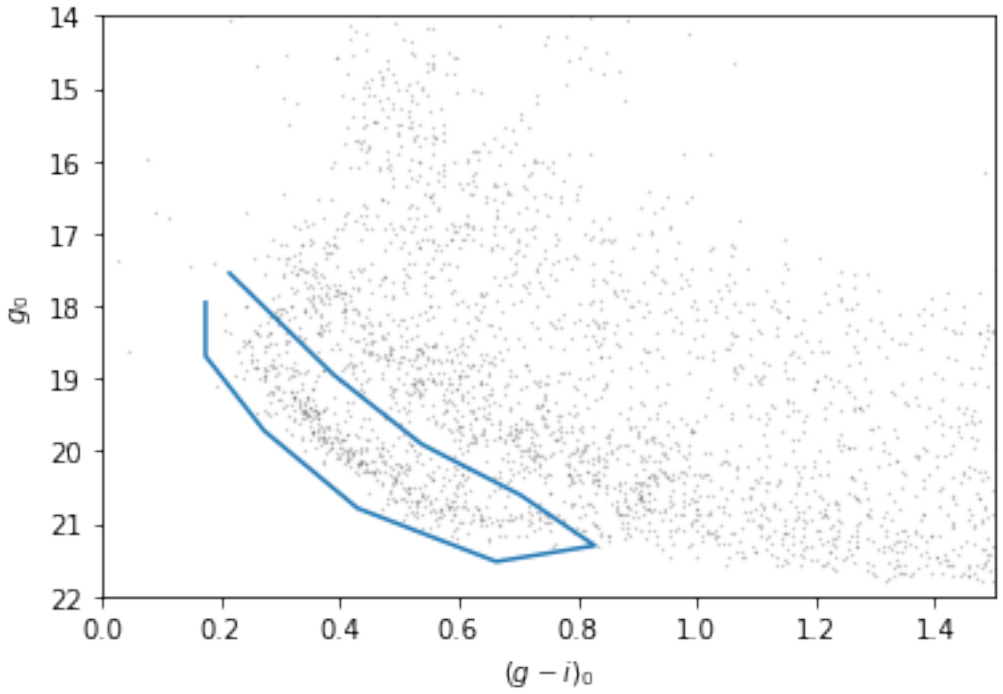
The next step is to convert the coordinates to a format we can use to plot them, which is a sequence of `x` coordinates and a sequence of `y` coordinates. The NumPy function `transpose` does what we want.

```
import numpy as np

xs, ys = np.transpose(coords)
xs, ys
```

```
(array([0.21505376, 0.38978495, 0.53763441, 0.70340502, 0.82885305,
        0.66308244, 0.43010753, 0.27329749, 0.17473118, 0.17473118]),
 array([17.5481972 , 18.94628403, 19.90286976, 20.60191317, 21.30095659,
        21.52170714, 20.78587196, 19.71891096, 18.68874172, 17.95290655]))
```

To display the polygon, we'll draw the figure again and use `plt.plot` to draw the polygon.

```
plot_cmd(photo_table)
plt.plot(xs, ys);
```



If it looks like your polygon does a good job surrounding the overdense area, go on to the next section. Otherwise you can try again.

If you want a polygon with more points (or fewer), you can change the argument to `ginput`.

The polygon does not have to be "closed". When we use this polygon in the next section, the last and first points will be connected by a straight line.

6.6 Which points are in the polygon?

Matplotlib provides a `Path` object that we can use to check which points fall in the polygon we selected.

Here's how we make a `Path` using a list of coordinates.

```
from matplotlib.path import Path

path = Path(coords)
path
```

```
Path(array([[ 0.21505376, 17.5481972 ],
       [ 0.38978495, 18.94628403],
       [ 0.53763441, 19.90286976],
       [ 0.70340502, 20.60191317],
       [ 0.82885305, 21.30095659],
       [ 0.66308244, 21.52170714],
       [ 0.43010753, 20.78587196],
       [ 0.27329749, 19.71891096],
       [ 0.17473118, 18.68874172],
       [ 0.17473118, 17.95290655]]), None)
```

`Path` provides `contains_points`, which figures out which points are inside the polygon.

To test it, we'll create a list with two points, one inside the polygon and one outside.

```
points = [(0.4, 20),
          (0.4, 30)]
```

Now we can make sure `contains_points` does what we expect.

```
inside = path.contains_points(points)
inside
```

```
array([ True, False])
```

The result is an array of Boolean values.

We are almost ready to select stars whose photometry data falls in this polygon. But first we need to do some data cleaning.

6.7 Reloading the data

Now we need to combine the photometry data with the list of candidate stars we identified in a previous notebook. The following cell downloads it:

```
import os
from wget import download

filename = 'gd1_candidates.hdf5'
filepath = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(filepath+filename))
```

```
import pandas as pd

candidate_df = pd.read_hdf(filename, 'candidate_df')
```

`candidate_df` is the Pandas DataFrame that contains the results from Notebook XX, which selects stars likely to be in GD-1 based on proper motion. It also includes position and proper motion transformed to the ICRS frame.

6.8 Merging photometry data

Before we select stars based on photometry data, we have to solve two problems:

1. We only have Pan-STARRS data for some stars in `candidate_df`.
2. Even for the stars where we have Pan-STARRS data in `photo_table`, some photometry data is missing.

We will solve these problems in two step:

1. We'll merge the data from `candidate_df` and `photo_table` into a single Pandas `DataFrame`.
2. We'll use Pandas functions to deal with missing data.

`candidate_df` is already a `DataFrame`, but `results` is an Astropy `Table`. Let's convert it to Pandas:

```
photo_df = photo_table.to_pandas()

for colname in photo_df.columns:
    print(colname)
```

```
source_id
g_mean_psf_mag
i_mean_psf_mag
```

Now we want to combine `candidate_df` and `photo_df` into a single table, using `source_id` to match up the rows.

You might recognize this task; it's the same as the JOIN operation in ADQL/SQL.

Pandas provides a function called `merge` that does what we want. Here's how we use it.

```
merged = pd.merge(candidate_df,
                  photo_df,
                  on='source_id',
                  how='left')
merged.head()
```

```
            source_id          ra        dec      pmra      pmdec  parallax  \
0  635559124339440000  137.586717  19.196544 -3.770522 -12.490482  0.791393
1  635860218726658176  138.518707  19.092339 -5.941679 -11.346409  0.307456
2  635674126383965568  138.842874  19.031798 -3.897001 -12.702780  0.779463
3  635535454774983040  137.837752  18.864007 -4.335041 -14.492309  0.314514
4  635497276810313600  138.044516  19.009471 -7.172931 -12.291499  0.425404

   parallax_error  radial_velocity       phi1      phi2   pm_phi1   pm_phi2  \
0        0.271754              NaN -59.630489 -1.216485 -7.361363 -0.592633
1        0.199466              NaN -59.247330 -2.016078 -7.527126  1.748779
2        0.223692              NaN -59.133391 -2.306901 -7.560608 -0.741800
3        0.102775              NaN -59.785300 -1.594569 -9.357536 -1.218492
```

(continues on next page)

(continued from previous page)

```
4        0.337689              NaN -59.557744 -1.682147 -9.000831  2.334407

   g_mean_psf_mag  i_mean_psf_mag
0             NaN             NaN
1         17.8978        17.517401
2         19.2873        17.678101
3         16.9238        16.478100
4         19.9242        18.334000
```

The first argument is the "left" table, the second argument is the "right" table, and the keyword argument `on='source_id'` specifies a column to use to match up the rows.

The argument `how='left'` means that the result should have all rows from the left table, even if some of them don't match up with a row in the right table.

If you are interested in the other options for `how`, you can read the documentation of `merge`.

You can also do different types of join in ADQL/SQL; you can read about that here.

The result is a `DataFrame` that contains the same number of rows as `candidate_df`.

```
len(candidate_df), len(photo_df), len(merged)
```

```
(7346, 3724, 7346)
```

And all columns from both tables.

```
for colname in merged.columns:
    print(colname)
```

```
source_id
ra
dec
pmra
pmdec
parallax
parallax_error
radial_velocity
phi1
phi2
pm_phi1
pm_phi2
g_mean_psf_mag
i_mean_psf_mag
```

Detail You might notice that Pandas also provides a function called `join`; it does almost the same thing, but the interface is slightly different. We think `merge` is a little easier to use, so that's what we chose. It's also more consistent with JOIN in SQL, so if you learn how to use `pd.merge`, you are also learning how to use SQL JOIN.

Also, someone might ask why we have to use Pandas to do this join; why didn't we do it in ADQL. The answer is that we could have done that, but since we already have the data we need, we should probably do the computation locally rather than make another round trip to the Gaia server.

6.9 Missing data

Let's add columns to the merged table for magnitude and color.

```
merged['mag'] = merged['g_mean_psf_mag']
merged['color'] = merged['g_mean_psf_mag'] - merged['i_mean_psf_mag']
```

These columns contain the special value `NaN` where we are missing data.

We can use `notnull` to see which rows contain value data, that is, not null values.

```
merged['color'].notnull()
```

```
0       False
1        True
2        True
3        True
4        True
        ...
7341     True
7342    False
7343    False
7344     True
7345    False
Name: color, Length: 7346, dtype: bool
```

And `sum` to count the number of valid values.

```
merged['color'].notnull().sum()
```

```
3724
```

For scientific purposes, it's not obvious what we should do with candidate stars if we don't have photometry data. Should we give them the benefit of the doubt or leave them out?

In part the answer depends on the goal: are we trying to identify more stars that might be in GD-1, or a smaller set of stars that have higher probability?

In the next section, we'll leave them out, but you can experiment with the alternative.

6.10 Selecting based on photometry

Now let's see how many of these points are inside the polygon we chose.

We can use a list of column names to select `color` and `mag`.

```
points = merged[['color', 'mag']]
points.head()
```

```
    color      mag
0     NaN      NaN
1  0.3804  17.8978
2  1.6092  19.2873
3  0.4457  16.9238
4  1.5902  19.9242
```

The result is a `DataFrame` that can be treated as a sequence of coordinates, so we can pass it to `contains_points`:

```
inside = path.contains_points(points)
inside
```

```
array([False, False, False, ..., False, False, False])
```

The result is a Boolean array. We can use `sum` to see how many stars fall in the polygon.

```
inside.sum()
```

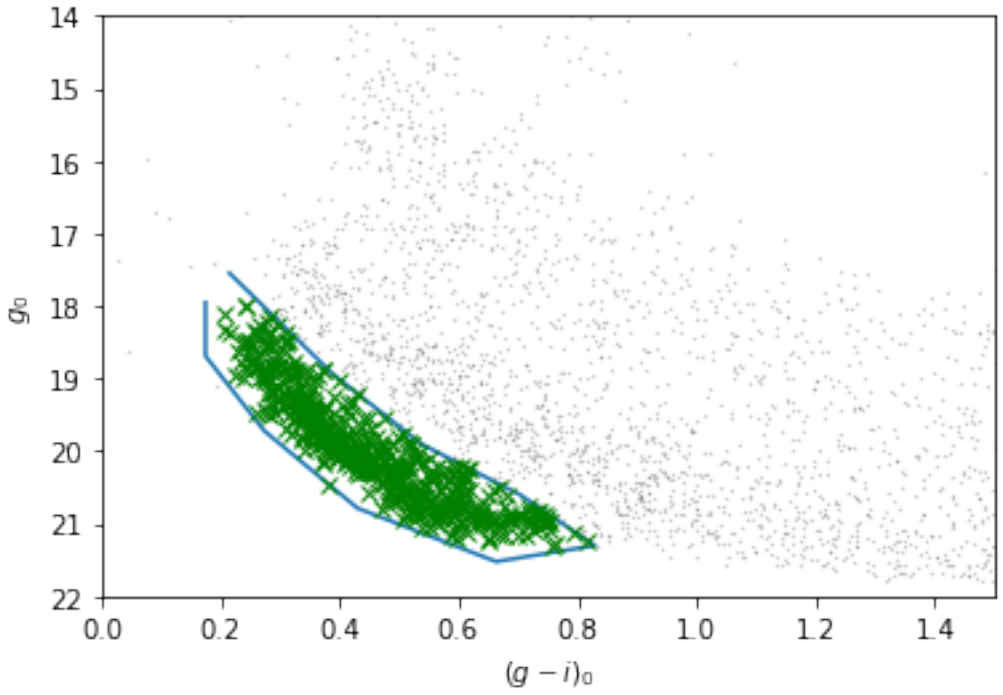
```
496
```

Now we can use `inside` as a mask to select stars that fall inside the polygon.

```
selected = merged[inside]
```

Let's make a color-magnitude plot one more time, highlighting the selected stars with green `x` marks.

```
plot_cmd(photo_table)
plt.plot(xs, ys)

plt.plot(selected['color'], selected['mag'], 'gx');
```



It looks like the selected stars are, in fact, inside the polygon, which means they have photometry data consistent with GD-1.

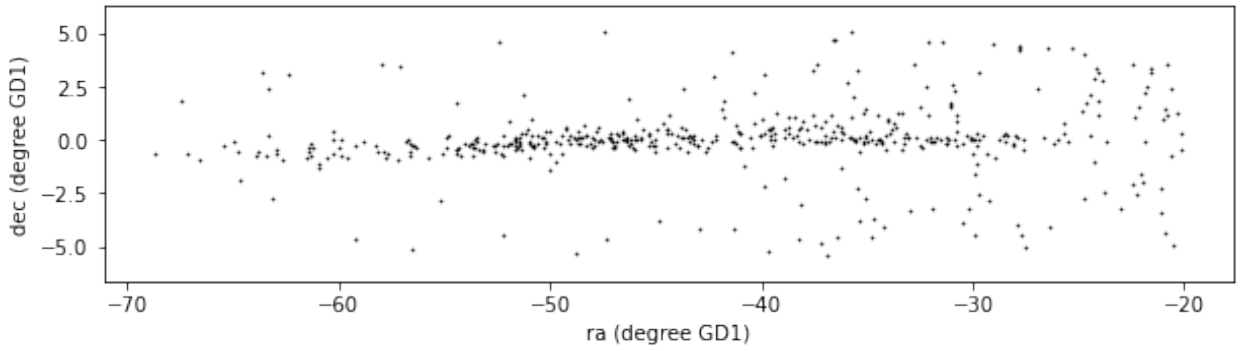
Finally, we can plot the coordinates of the selected stars:

```
plt.figure(figsize=(10,2.5))
```

(continues on next page)

(continued from previous page)

```
x = selected['phi1']
y = selected['phi2']

plt.plot(x, y, 'ko', markersize=0.7, alpha=0.9)

plt.xlabel('ra (degree GD1)')
plt.ylabel('dec (degree GD1)')

plt.axis('equal');
```



This example includes two new Matplotlib commands:

- `figure` creates the figure. In previous examples, we didn't have to use this function; the figure was created automatically. But when we call it explicitly, we can provide arguments like `figsize`, which sets the size of the figure.
- `axis` with the parameter `equal` sets up the axes so a unit is the same size along the `x` and `y` axes.

In an example like this, where `x` and `y` represent coordinates in space, equal axes ensures that the distance between points is represented accurately.

6.11 Write the data

Let's write the merged DataFrame to a file.

```
filename = 'gd1_merged.hdf5'

merged.to_hdf(filename, 'merged')
selected.to_hdf(filename, 'selected')
```

```
!ls -lh gd1_merged.hdf5
```

```
-rw-rw-r-- 1 downey downey 2.0M Oct 19 17:21 gd1_merged.hdf5
```

If you are using Windows, `ls` might not work; in that case, try:

```
!dir gd1_merged.hdf5
```

6.12 Save the polygon

Reproducibile research is "the idea that ... the full computational environment used to produce the results in the paper such as the code, data, etc. can be used to reproduce the results and create new work based on the research."

This Jupyter notebook is an example of reproducible research because it contains all of the code needed to reproduce the results, including the database queries that download the data and and analysis.

However, when we used `ginput` to define a polygon by hand, we introduced a non-reproducible element to the analysis. If someone running this notebook chooses a different polygon, they will get different results. So it is important to record the polygon we chose as part of the data analysis pipeline.

Since `coords` is a NumPy array, we can't use `to_hdf` to save it in a file. But we can convert it to a Pandas `DataFrame` and save that.

As an alternative, we could use PyTables, which is the library Pandas uses to read and write files. It is a powerful library, but not easy to use directly. So let's take advantage of Pandas.

```
coords_df = pd.DataFrame(coords)
```

```
filename = 'gd1_polygon.hdf5'
coords_df.to_hdf(filename, 'coords_df')
```

We can read it back like this.

```
coords2_df = pd.read_hdf(filename, 'coords_df')
coords2 = coords2_df.to_numpy()
```

And verify that the data we read back is the same.

```
np.all(coords2 == coords)
```

```
True
```

6.13 Summary

In this notebook, we worked with two datasets: the list of candidate stars from Gaia and the photometry data from Pan-STARRS.

We drew a color-magnitude diagram and used it to identify stars we think are likely to be in GD-1.

Then we used a Pandas `merge` operation to combine the data into a single `DataFrame`.

6.14 Best practices

- If you want to perform something like a database `JOIN` operation with data that is in a Pandas `DataFrame`, you can use the `join` or `merge` function. In many cases, `merge` is easier to use because the arguments are more like SQL.
- Use Matplotlib options to control the size and aspect ratio of figures to make them easier to interpret. In this example, we scaled the axes so the size of a degree is equal along both axes.
- Matplotlib also provides operations for working with points, polygons, and other geometric entities, so it's not just for making figures.

- Be sure to record every element of the data analysis pipeline that would be needed to replicate the results.

CHAPTER

SEVEN

CHAPTER 7

This is the seventh in a series of notebooks related to astronomy data.

As a continuing example, we will replicate part of the analysis in a recent paper, "Off the beaten path: Gaia reveals GD-1 stars outside of the main stream" by Adrian M. Price-Whelan and Ana Bonaca.

In the previous notebook we selected photometry data from Pan-STARRS and used it to identify stars we think are likely to be in GD-1

In this notebook, we'll take the results from previous lessons and use them to make a figure that tells a compelling scientific story.

7.1 Outline

Here are the steps in this notebook:

1. Starting with the figure from the previous notebook, we'll add annotations to present the results more clearly.
2. The we'll see several ways to customize figures to make them more appealing and effective.
3. Finally, we'll see how to make a figure with multiple panels or subplots.

After completing this lesson, you should be able to

- Design a figure that tells a compelling story.
- Use Matplotlib features to customize the appearance of figures.
- Generate a figure with multiple subplots.

7.2 Installing libraries

If you are running this notebook on Colab, you can run the following cell to install Astroquery and a the other libraries we'll use.

If you are running this notebook on your own computer, you might have to install these libraries yourself.

If you are using this notebook as part of a Carpentries workshop, you should have received setup instructions.

TODO: Add a link to the instructions.

```
# If we're running on Colab, install libraries

import sys
IN_COLAB = 'google.colab' in sys.modules
```

(continues on next page)

(continued from previous page)

```
if IN_COLAB:
    !pip install astroquery astro-gala pyia python-wget
```

7.3 Making Figures That Tell a Story

So far the figure we've made have been "quick and dirty". Mostly we have used Matplotlib's default style, although we have adjusted a few parameters, like `markersize` and `alpha`, to improve legibility.

Now that the analysis is done, it's time to think more about:

1. Making professional-looking figures that are ready for publication, and
2. Making figures that communicate a scientific result clearly and compellingly.

Not necessarily in that order.

Let's start by reviewing Figure 1 from the original paper. We've seen the individual panels, but now let's look at the whole thing, along with the caption:

Exercise: Think about the following questions:

1. What is the primary scientific result of this work?
2. What story is this figure telling?
3. In the design of this figure, can you identify 1-2 choices the authors made that you think are effective? Think about big-picture elements, like the number of panels and how they are arranged, as well as details like the choice of typeface.
4. Can you identify 1-2 elements that could be improved, or that you might have done differently?

Some topics that might come up in this discussion:

1. The primary result is that the multiple stages of selection make it possible to separate likely candidates from the background more effectively than in previous work, which makes it possible to see the structure of GD-1 in "unprecedented detail".
2. The figure documents the selection process as a sequence of steps. Reading right-to-left, top-to-bottom, we see selection based on proper motion, the results of the first selection, selection based on color and magnitude, and the results of the second selection. So this figure documents the methodology and presents the primary result.
3. It's mostly black and white, with minimal use of color, so it will work well in print. The annotations in the bottom left panel guide the reader to the most important results. It contains enough technical detail for a professional audience, but most of it is also comprehensible to a more general audience. The two left panels have the same dimensions and their axes are aligned.
4. Since the panels represent a sequence, it might be better to arrange them left-to-right. The placement and size of the axis labels could be tweaked. The entire figure could be a little bigger to match the width and proportion of the caption. The top left panel has unnused white space (but that leaves space for the annotations in the bottom left).

7.4 Plotting GD-1

Let's start with the panel in the lower left. The following cell reloads the data.

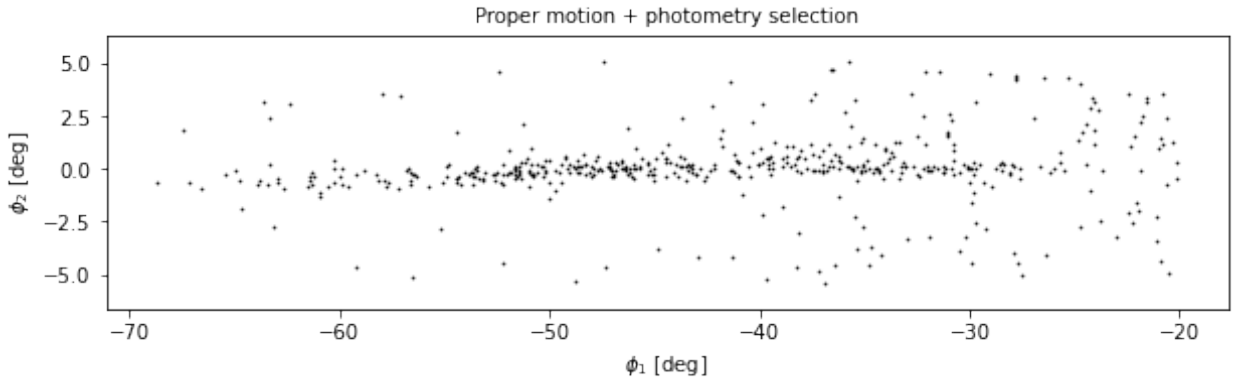
```
import os
from wget import download

filename = 'gd1_merged.hdf5'
path = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(path+filename))
```

```
import pandas as pd

selected = pd.read_hdf(filename, 'selected')
```

```
import matplotlib.pyplot as plt

def plot_second_selection(df):
    x = df['phi1']
    y = df['phi2']

    plt.plot(x, y, 'ko', markersize=0.7, alpha=0.9)

    plt.xlabel('$\phi_1$ [deg]')
    plt.ylabel('$\phi_2$ [deg]')
    plt.title('Proper motion + photometry selection', fontsize='medium')

    plt.axis('equal')
```

And here's what it looks like.

```
plt.figure(figsize=(10,2.5))
plot_second_selection(selected)
```



7.5 Annotations

The figure in the paper uses three other features to present the results more clearly and compellingly:

- A vertical dashed line to distinguish the previously undetected region of GD-1,
- A label that identifies the new region, and
- Several annotations that combine text and arrows to identify features of GD-1.

As an exercise, choose any or all of these features and add them to the figure:

- To draw vertical lines, see `plt.vlines` and `plt.axvline`.
- To add text, see `plt.text`.
- To add an annotation with text and an arrow, see plt.annotate.

And here is some additional information about text and arrows.

```
# Solution

# plt.axvline(-55, ls='--', color='gray',
#             alpha=0.4, dashes=(6,4), lw=2)
# plt.text(-60, 5.5, 'Previously\nundetected',
#          fontsize='small', ha='right', va='top');

# arrowprops=dict(color='gray', shrink=0.05, width=1.5,
#                 headwidth=6, headlength=8, alpha=0.4)

# plt.annotate('Spur', xy=(-33, 2), xytext=(-35, 5.5),
#              arrowprops=arrowprops,
#              fontsize='small')

# plt.annotate('Gap', xy=(-22, -1), xytext=(-25, -5.5),
#              arrowprops=arrowprops,
#              fontsize='small')
```

7.6 Customization

Matplotlib provides a default style that determines things like the colors of lines, the placement of labels and ticks on the axes, and many other properties.

There are several ways to override these defaults and customize your figures:

- To customize only the current figure, you can call functions like `tick_params`, which we'll demonstrate below.
- To customize all figures in a notebook, you use `rcParams`.
- To override more than a few defaults at the same time, you can use a style sheet.

As a simple example, notice that Matplotlib puts ticks on the outside of the figures by default, and only on the left and bottom sides of the axes.

To change this behavior, you can use `gca()` to get the current axes and `tick_params` to change the settings.

Here's how you can put the ticks on the inside of the figure:

```
plt.gca().tick_params(direction='in')
```

Exercise: Read the documentation of `tick_params` and use it to put ticks on the top and right sides of the axes.

```
# Solution

# plt.gca().tick_params(top=True, right=True)
```

7.7 rcParams

If you want to make a customization that applies to all figures in a notebook, you can use `rcParams`.

Here's an example that reads the current font size from `rcParams`:

```
plt.rcParams['font.size']
```

```
10.0
```

And sets it to a new value:

```
plt.rcParams['font.size'] = 14
```

Exercise: Plot the previous figure again, and see what font sizes have changed. Look up any other element of `rcParams`, change its value, and check the effect on the figure.

If you find yourself making the same customizations in several notebooks, you can put changes to `rcParams` in a `matplotlibrc` file, which you can read about here.

7.8 Style sheets

The `matplotlibrc` file is read when you import Matplotlib, so it is not easy to switch from one set of options to another.

The solution to this problem is style sheets, which you can read about here.

Matplotlib provides a set of predefined style sheets, or you can make your own.

The following cell displays a list of style sheets installed on your system.

```
plt.style.available
```

```
['Solarize_Light2',
 '_classic_test_patch',
 'bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn',
 'seaborn-bright',
 'seaborn-colorblind',
```

(continues on next page)

(continued from previous page)

```
 'seaborn-dark',
 'seaborn-dark-palette',
 'seaborn-darkgrid',
 'seaborn-deep',
 'seaborn-muted',
 'seaborn-notebook',
 'seaborn-paper',
 'seaborn-pastel',
 'seaborn-poster',
 'seaborn-talk',
 'seaborn-ticks',
 'seaborn-white',
 'seaborn-whitegrid',
 'tableau-colorblind10']
```

Note that `seaborn-paper`, `seaborn-talk` and `seaborn-poster` are particularly intended to prepare versions of a figure with text sizes and other features that work well in papers, talks, and posters.

To use any of these style sheets, run `plt.style.use` like this:

```
plt.style.use('fivethirtyeight')
```

The style sheet you choose will affect the appearance of all figures you plot after calling `use`, unless you override any of the options or call `use` again.

Exercise: Choose one of the styles on the list and select it by calling `use`. Then go back and plot one of the figures above and see what effect it has.

If you can't find a style sheet that's exactly what you want, you can make your own. This repository includes a style sheet called `az-paper-twocol.mplstyle`, with customizations chosen by Azalee Bostroem for publication in astronomy journals.

The following cell downloads the style sheet.

```
import os

filename = 'az-paper-twocol.mplstyle'
path = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(path+filename))
```

You can use it like this:

```
plt.style.use('./az-paper-twocol.mplstyle')
```

The prefix `./` tells Matplotlib to look for the file in the current directory.

As an alternative, you can install a style sheet for your own use by putting it in your configuration directory. To find out where that is, you can run the following command:

```
import matplotlib as mpl

mpl.get_configdir()
```

7.9 LaTeX fonts

When you include mathematical expressions in titles, labels, and annotations, Matplotlib uses `mathtext` to typeset them. `mathtext` uses the same syntax as LaTeX, but it provides only a subset of its features.

If you need features that are not provided by `mathtext`, or you prefer the way LaTeX typesets mathematical expressions, you can customize Matplotlib to use LaTeX.

In `matplotlibrc` or in a style sheet, you can add the following line:

```
text.usetex        : true
```

Or in a notebook you can run the following code.

```
plt.rcParams['text.usetex'] = True
```


```
plt.rcParams['text.usetex'] = True
```


If you go back and draw the figure again, you should see the difference.

If you get an error message like

```
LaTeX Error: File `type1cm.sty' not found.
```

You might have to install a package that contains the fonts LaTeX needs. On some systems, the packages `texlive-latex-extra` or `cm-super` might be what you need. See here for more help with this.

In case you are curious, `cm` stands for Computer Modern, the font LaTeX uses to typeset math.

7.10 Multiple panels

So far we've been working with one figure at a time, but the figure we are replicating contains multiple panels, also known as "subplots".

Confusingly, Matplotlib provides *three* functions for making figures like this: `subplot`, `subplots`, and `subplot2grid`.

- `subplot` is simple and similar to MATLAB, so if you are familiar with that interface, you might like `subplot`
- `subplots` is more object-oriented, which some people prefer.
- `subplot2grid` is most convenient if you want to control the relative sizes of the subplots.

So we'll use `subplot2grid`.

All of these functions are easier to use if we put the code that generates each panel in a function.

7.11 Upper right

To make the panel in the upper right, we have to reload `centerline`.

```
import os

filename = 'gd1_dataframe.hdf5'
path = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(path+filename))
```

```
import pandas as pd

centerline = pd.read_hdf(filename, 'centerline')
```

And define the coordinates of the rectangle we selected.

```
pm1_min = -8.9
pm1_max = -6.9
pm2_min = -2.2
pm2_max =  1.0

pm1_rect = [pm1_min, pm1_min, pm1_max, pm1_max]
pm2_rect = [pm2_min, pm2_max, pm2_max, pm2_min]
```

To plot this rectangle, we'll use a feature we have not seen before: `Polygon`, which is provided by Matplotlib.

To create a `Polygon`, we have to put the coordinates in an array with `x` values in the first column and `y` values in the second column.

```
import numpy as np

vertices = np.transpose([pm1_rect, pm2_rect])
vertices
```

```
array([[-8.9, -2.2],
       [-8.9,  1. ],
       [-6.9,  1. ],
       [-6.9, -2.2]])
```

The following function takes a `DataFrame` as a parameter, plots the proper motion for each star, and adds a shaded `Polygon` to show the region we selected.

```
from matplotlib.patches import Polygon

def plot_proper_motion(df):
    pm1 = df['pm_phi1']
    pm2 = df['pm_phi2']

    plt.plot(pm1, pm2, 'ko', markersize=0.3, alpha=0.3)

    poly = Polygon(vertices, closed=True,
                   facecolor='C1', alpha=0.4)
    plt.gca().add_patch(poly)
```

(continues on next page)

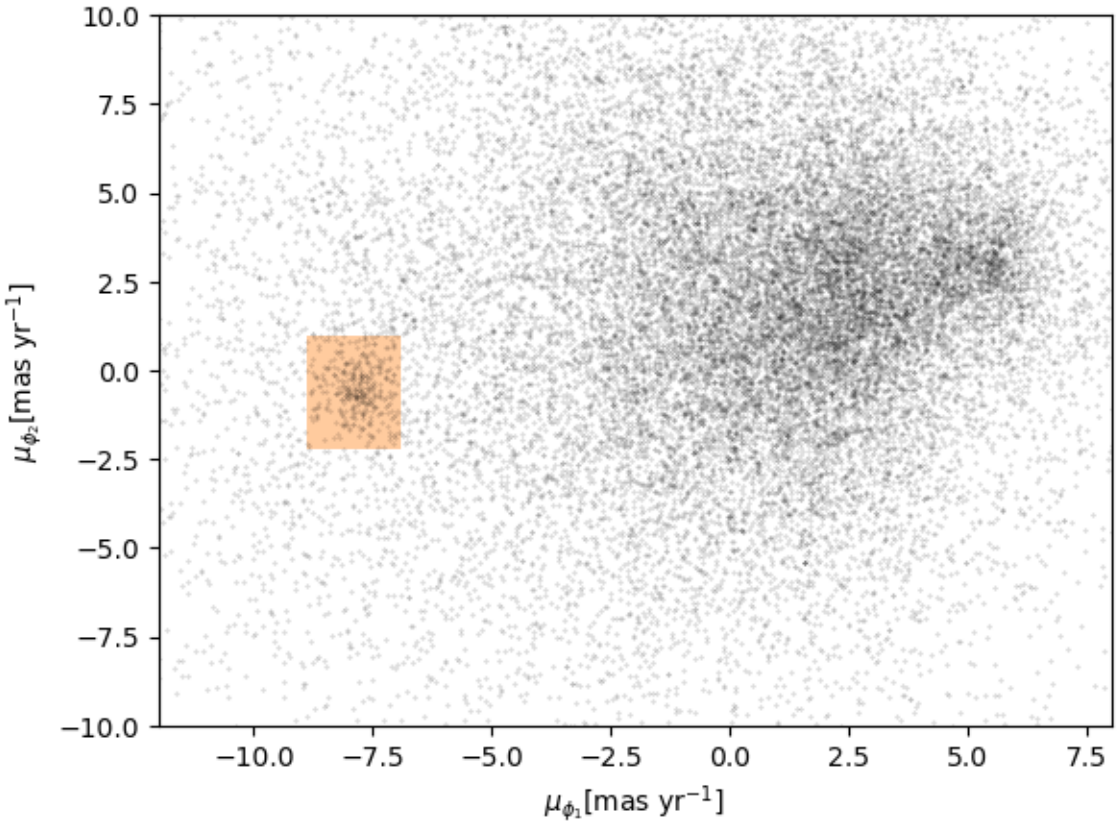
(continued from previous page)

```
    plt.xlabel('$\mu_{\phi_1} [\mathrm{mas~yr}^{-1}]$')
    plt.ylabel('$\mu_{\phi_2} [\mathrm{mas~yr}^{-1}]$')

    plt.xlim(-12, 8)
    plt.ylim(-10, 10)
```

Notice that `add_patch` is like `invert_yaxis`; in order to call it, we have to use `gca` to get the current axes.

Here's what the new version of the figure looks like. We've changed the labels on the axes to be consistent with the paper.

```
plt.rcParams['text.usetex'] = False
plt.style.use('default')

plot_proper_motion(centerline)
```



7.12 Upper left

Now let's work on the panel in the upper left. We have to reload `candidates`.

```
import os

filename = 'gd1_candidates.hdf5'
path = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(path+filename))
```

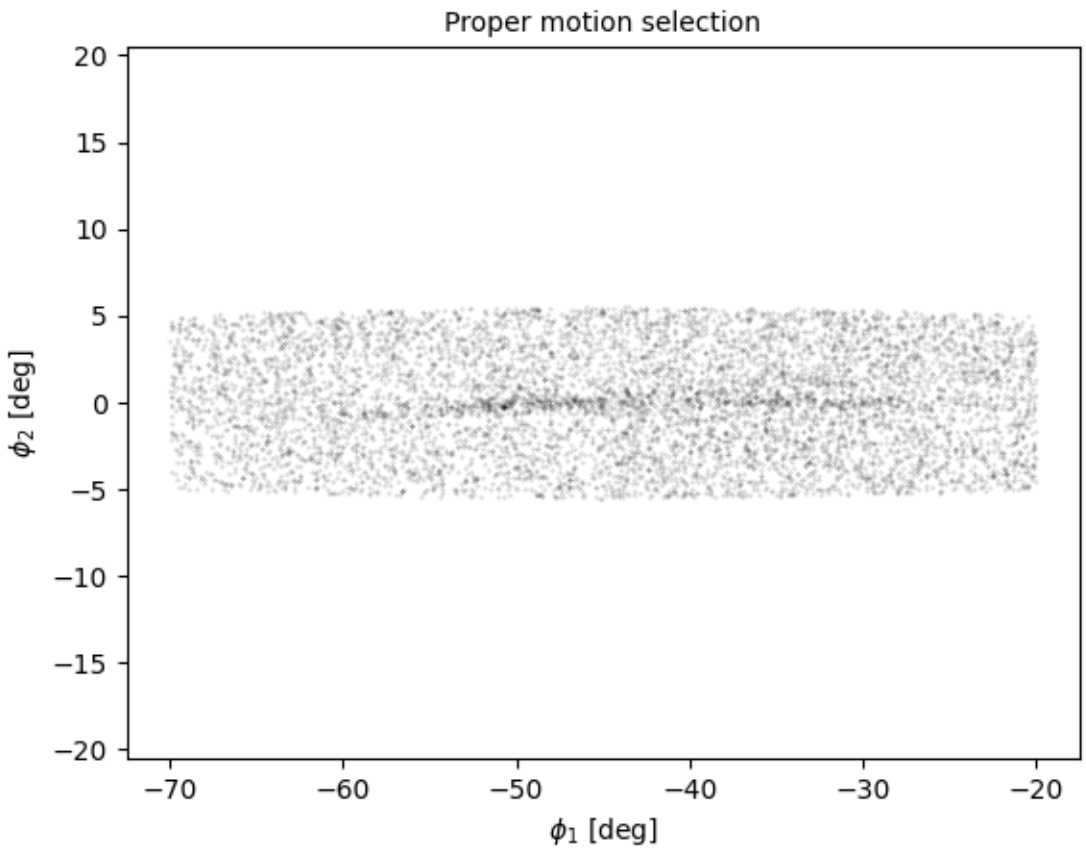
```
import pandas as pd

filename = 'gd1_candidates.hdf5'

candidate_df = pd.read_hdf(filename, 'candidate_df')
```

Here's a function that takes a `DataFrame` of candidate stars and plots their positions in GD-1 coordindates.

```
def plot_first_selection(df):
    x = df['phi1']
    y = df['phi2']

    plt.plot(x, y, 'ko', markersize=0.3, alpha=0.3)

    plt.xlabel('$\phi_1$ [deg]')
    plt.ylabel('$\phi_2$ [deg]')
    plt.title('Proper motion selection', fontsize='medium')

    plt.axis('equal')
```

And here's what it looks like.

```
plot_first_selection(candidate_df)
```



7.13 Lower right

For the figure in the lower right, we need to reload the merged `DataFrame`, which contains data from Gaia and photometry data from Pan-STARRS.

```
import pandas as pd

filename = 'gd1_merged.hdf5'

merged = pd.read_hdf(filename, 'merged')
```

From the previous notebook, here's the function that plots the color-magnitude diagram.

```
import matplotlib.pyplot as plt

def plot_cmd(table):
    """Plot a color magnitude diagram.

    table: Table or DataFrame with photometry data
    """
    y = table['g_mean_psf_mag']
    x = table['g_mean_psf_mag'] - table['i_mean_psf_mag']

    plt.plot(x, y, 'ko', markersize=0.3, alpha=0.3)
```

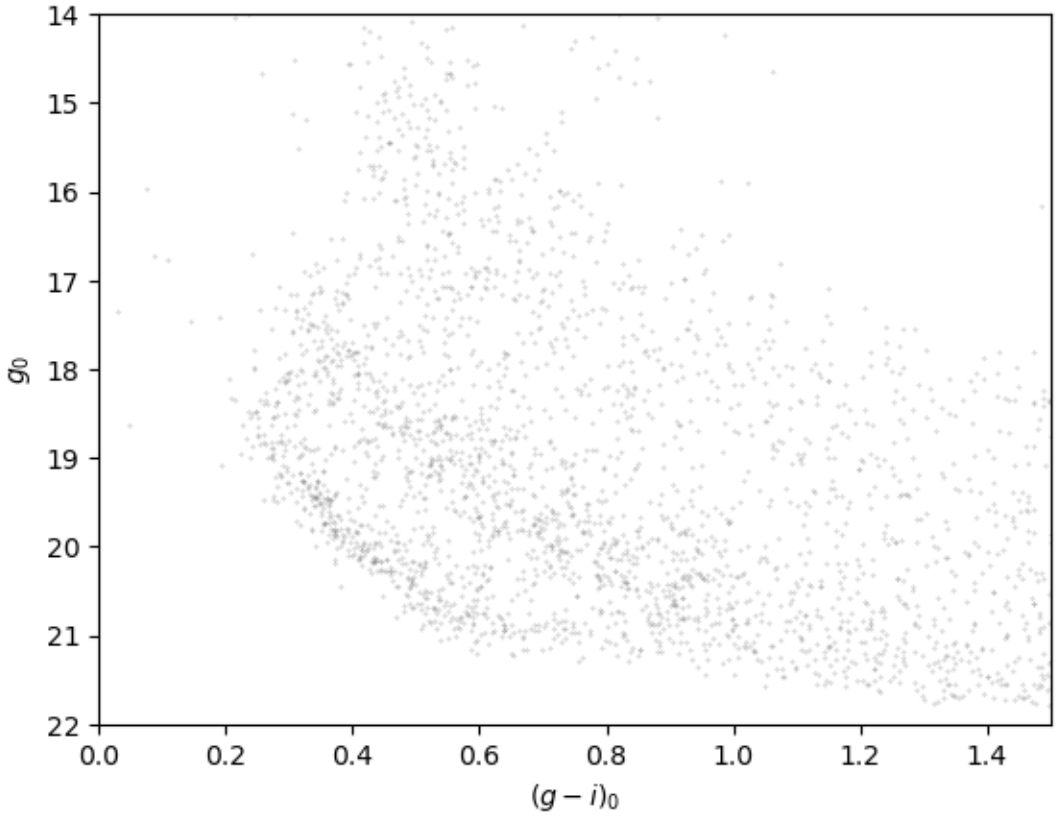
(continues on next page)

(continued from previous page)

```
    plt.xlim([0, 1.5])
    plt.ylim([14, 22])
    plt.gca().invert_yaxis()

    plt.ylabel('$g_0$')
    plt.xlabel('$(g-i)_0$')
```

And here's what it looks like.

```
plot_cmd(merged)
```



Exercise: Add a few lines to `plot_cmd` to show the Polygon we selected as a shaded area.

Run these cells to get the polygon coordinates we saved in the previous notebook.

```
import os

filename = 'gd1_polygon.hdf5'
path = 'https://github.com/AllenDowney/AstronomicalData/raw/main/data/'

if not os.path.exists(filename):
    print(download(path+filename))
```

```
coords_df = pd.read_hdf(filename, 'coords_df')
coords = coords_df.to_numpy()
coords
```

```
array([[ 0.21505376, 17.5481972 ],
       [ 0.38978495, 18.94628403],
       [ 0.53763441, 19.90286976],
       [ 0.70340502, 20.60191317],
       [ 0.82885305, 21.30095659],
       [ 0.66308244, 21.52170714],
       [ 0.43010753, 20.78587196],
       [ 0.27329749, 19.71891096],
       [ 0.17473118, 18.68874172],
       [ 0.17473118, 17.95290655]])
```

```
# Solution

#poly = Polygon(coords, closed=True,
#                facecolor='C1', alpha=0.4)
#plt.gca().add_patch(poly)
```

7.14 Subplots

Now we're ready to put it all together. To make a figure with four subplots, we'll use `subplot2grid`, which requires two arguments:

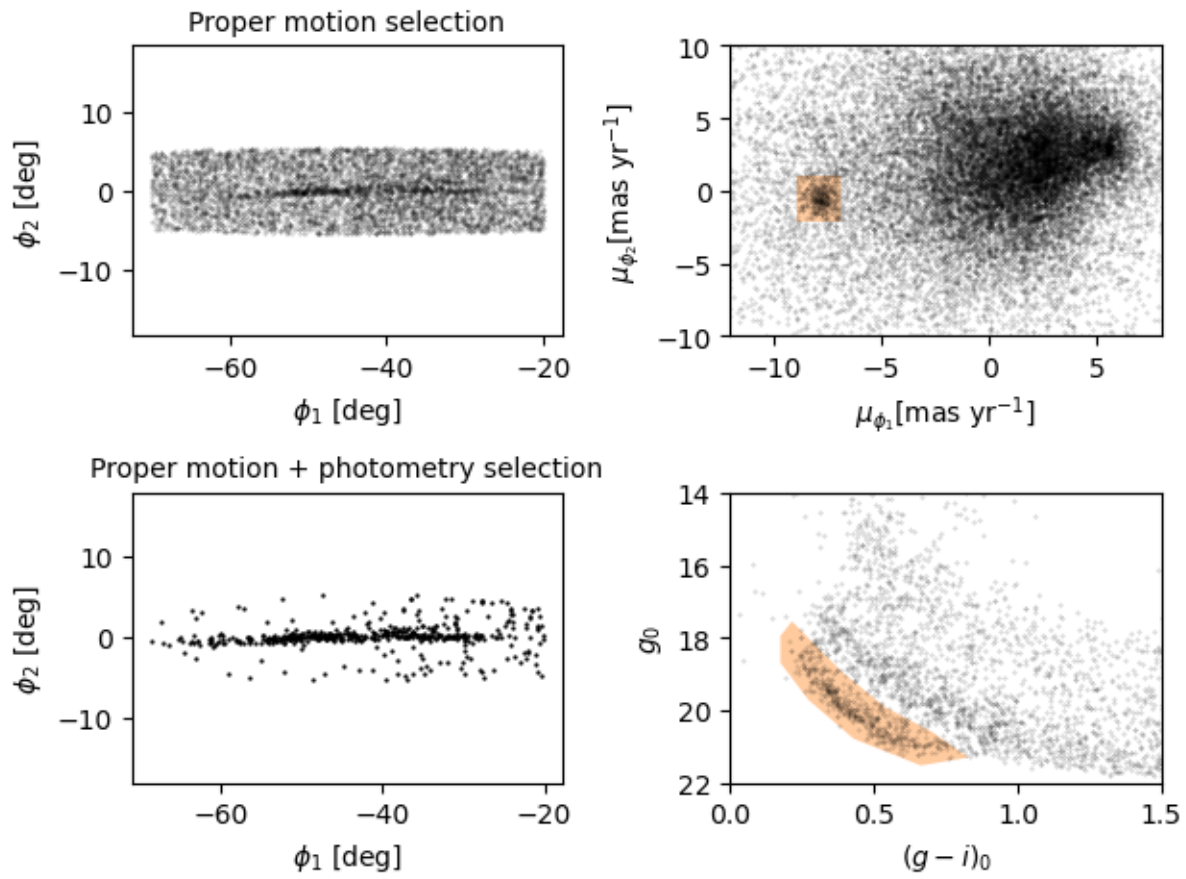
- `shape`, which is a tuple with the number of rows and columns in the grid, and
- `loc`, which is a tuple identifying the location in the grid we're about to fill.

In this example, `shape` is `(2, 2)` to create two rows and two columns.

For the first panel, `loc` is `(0, 0)`, which indicates row 0 and column 0, which is the upper-left panel.

Here's how we use it to draw the four panels.

```
shape = (2, 2)
plt.subplot2grid(shape, (0, 0))
plot_first_selection(candidate_df)

plt.subplot2grid(shape, (0, 1))
plot_proper_motion(centerline)

plt.subplot2grid(shape, (1, 0))
plot_second_selection(selected)

plt.subplot2grid(shape, (1, 1))
plot_cmd(merged)
poly = Polygon(coords, closed=True,
               facecolor='C1', alpha=0.4)
plt.gca().add_patch(poly)

plt.tight_layout()
```



We use `plt.tight_layout` at the end, which adjusts the sizes of the panels to make sure the titles and axis labels don't overlap.

Exercise: See what happens if you leave out `tight_layout`.

7.15 Adjusting proportions

In the previous figure, the panels are all the same size. To get a better view of GD-1, we'd like to stretch the panels on the left and compress the ones on the right.

To do that, we'll use the `colspan` argument to make a panel that spans multiple columns in the grid.

In the following example, `shape` is `(2, 4)`, which means 2 rows and 4 columns.

The panels on the left span three columns, so they are three times wider than the panels on the right.

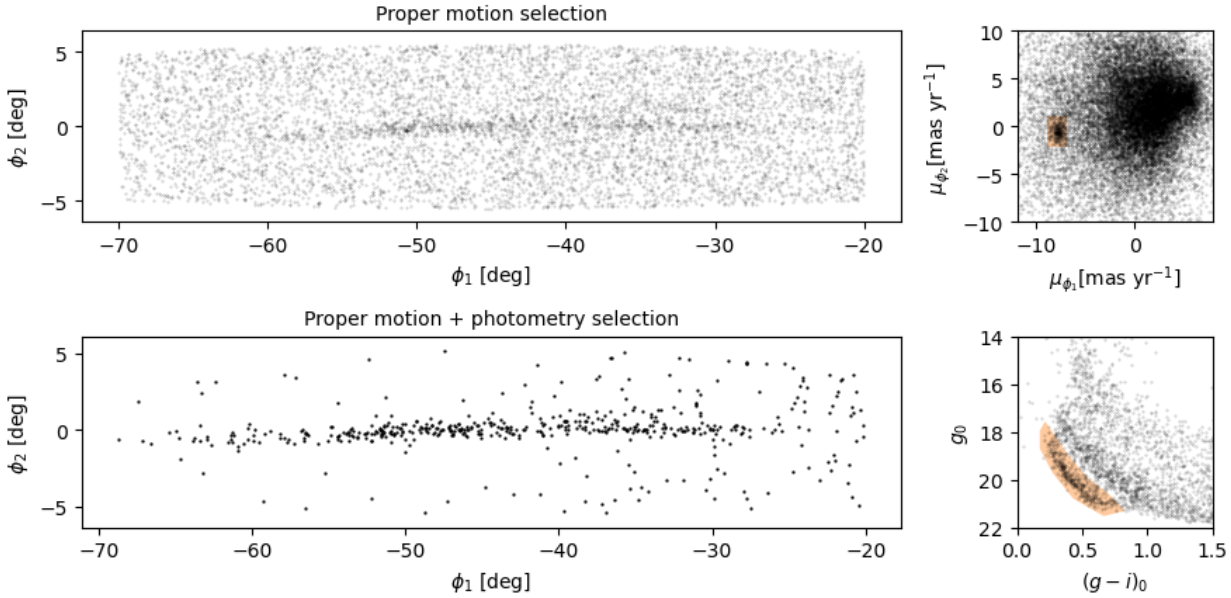
At the same time, we use `figsize` to adjust the aspect ratio of the whole figure.

```
plt.figure(figsize=(9, 4.5))

shape = (2, 4)
plt.subplot2grid(shape, (0, 0), colspan=3)
plot_first_selection(candidate_df)

plt.subplot2grid(shape, (0, 3))
plot_proper_motion(centerline)
```

(continues on next page)

(continued from previous page)

```
plt.subplot2grid(shape, (1, 0), colspan=3)
plot_second_selection(selected)

plt.subplot2grid(shape, (1, 3))
plot_cmd(merged)
poly = Polygon(coords, closed=True,
               facecolor='C1', alpha=0.4)
plt.gca().add_patch(poly)

plt.tight_layout()
```



This is looking more and more like the figure in the paper.

Exercise: In this example, the ratio of the widths of the panels is 3:1. How would you adjust it if you wanted the ratio to be 3:2?

7.16 Summary

In this notebook, we reverse-engineered the figure we've been replicating, identifying elements that seem effective and others that could be improved.

We explored features Matplotlib provides for adding annotations to figures – including text, lines, arrows, and polygons – and several ways to customize the appearance of figures. And we learned how to create figures that contain multiple panels.

7.17 Best practices

- The most effective figures focus on telling a single story clearly and compellingly.
- Consider using annotations to guide the readers attention to the most important elements of a figure.
- The default Matplotlib style generates good quality figures, but there are several ways you can override the defaults.
- If you find yourself making the same customizations on several projects, you might want to create your own style sheet.