1. Queries

Print to PDF

This is the first in a series of lessons about working with astronomical data.

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

Outline

This 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.

Query Language

In order to select data from a database, you have to compose a query, which is 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.

Using Jupyter

If you have not worked with Jupyter notebooks before, you might start with the tutorial on from Jupyter.org called "Try Classic Notebook", or this tutorial from DataQuest.

There are two environments you can use to write and run notebooks:

- "Jupyter Notebook" is the original, and
- "Jupyter Lab" is a newer environment with more features.

For these lessons, you can use either one.

If you are too impatient for the tutorials, here are the most important things to know:

1. Notebooks are made up of code cells and text cells (and a few other less common kinds). Code cells contain code; text cells, like this one, contain explanatory text written in Markdown.
2. To run a code cell, click the cell to select it and press Shift-Enter. The output of the code should appear below the cell.
3. In general, notebooks only run correctly if you run every code cell in order from top to bottom. If you run cells out of order, you are likely to get errors.
4. You can modify existing cells, but then you have to run them again to see the effect.
5. You can add new cells, but again, you have to be careful about the order you run them in.
6. If you have added or modified cells and the behavior of the notebook seems strange, you can restart the "kernel", which clears all of the variables and functions you have defined, and run the cells again from the beginning.

- If you are using Jupyter notebook, open the `Kernel` menu and select "Restart and Run All".
- In Jupyter Lab, open the `Kernel` menu and select "Restart Kernel and Run All Cells"
- In Colab, open the `Runtime` menu and select "Restart and run all"

Before you go on, you might want to explore the other menus and the toolbar to see what else you can do.

Connecting to Gaia

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

This import statement creates a TAP+ connection; TAP stands for "Table Access Protocol", which is a network protocol for sending queries to the database and getting back the results.

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 "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]
```

The following `for` loop prints the names of the tables.

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

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, not the contents of the table.

```
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 0x7f50edd2aeb0>
```

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
```

Columns

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

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

You can probably infer 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.panstarrs1_original_valid`. Use `load_table` to get the metadata for this table. How many columns are there and what are their names?

Writing queries

By now you might be wondering how we download these tables. 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, 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.

Also, the query is broken into multiple lines to make it more readable. This is a common style, but not required. Line breaks don't affect the behavior of the query.

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

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

```
<astroquery.utils.tap.model.job.Job at 0x7f50edd2adc0>
```

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

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

```
print(job)
```

```
<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
 parallax float64  mas                                                           Parallax
2
Jobid: None
Phase: COMPLETED
Owner: None
Output file: sync_20210315090602.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:

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

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

The `type` function indicates that the result is an Astropy 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*.

An Astropy `Table` 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`.

```
results
```

Table length=10

source_id	ra	dec	paralla
	deg	deg	ma
int64	float64	float64	float6
5887983246081387776	227.978818386372	-53.64996962450103	1.04931721633329
5887971250213117952	228.32280834041364	-53.66270726203726	0.29455652682279
5887991866047288704	228.1582047014091	-53.454724911639794	-0.57891799416692
5887968673232040832	228.07420888099884	-53.8064612895961	0.41030970779603
5887979844465854720	228.42547805195946	-53.48882284470035	-0.23379683441525
5887978607515442688	228.23831627636855	-53.56328249482688	-0.92521619567890
5887978298278520704	228.26015640396173	-53.607284412896476	-
5887995581231772928	228.12871598211902	-53.373625663608316	-0.33258182064393
5887982043490374016	227.985260087594	-53.683444499055575	0.02878111976456
5887982971205433856	227.89884570686218	-53.67430215342567	-

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?

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 in the next lesson.

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 these changes:

- It selects the first 3000 rows, so it is bigger than we should run synchronously.
- It selects two additional columns, `pmra` and `pmdec`, which are proper motions along the axes of `ra` and `dec`.
- It uses a new keyword, `WHERE`.

```
query2 = """SELECT
TOP 3000
source_id, ra, dec, pmra, pmdec, 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 download only the rows we need from the database.

We use `launch_job_async` to submit an asynchronous query.

```
job = Gaia.launch_job_async(query2)
job
```

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

```
<astroquery.utils.tap.model.job.Job at 0x7f50edd40f40>
```

And here are the results.

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

Table length=10

source_id	ra	dec	parallax
	deg	deg	mas
int64	float64	float64	float64
5895270396817359872	213.08433715252883	-56.64104701005694	2.041947005434917
5895272561481374080	213.2606587905109	-56.55044401535715	0.156934678951013
5895247410183786368	213.38479712976664	-56.97008551185148	-0.190175257425526
5895249226912448000	213.41587389088238	-56.849596577635786	--
5895261875598904576	213.5508930114549	-56.61037780154348	-0.294717223635292
5895258302187834624	213.87631129557286	-56.678537259039906	0.646843701528975
5895247444506644992	213.33215109206796	-56.975347759380995	0.39021549023428
5895259470417635968	213.78815034206346	-56.64585047451594	0.95337771078891
5895264899260932352	213.21521027193236	-56.78420864489118	--
5895265925746051584	213.17082359534547	-56.74540885107754	0.298691821510175

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

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 modified 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 test run faster, which reduces your development time.
- Launching test queries synchronously might make them start faster, too.

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

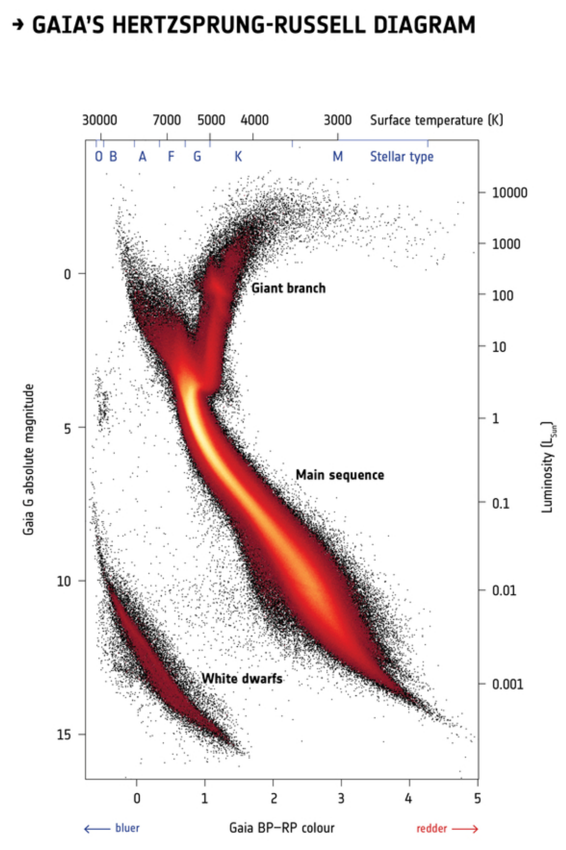
Finally, you can use `NOT` to invert the result of a comparison.

Exercise

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

`bp_rp` contains BP-RP color, which is the difference between two other columns, `phot_bp_mean_mag` and `phot_rp_mean_mag`. You can read about this variable here.

This Hertzsprung-Russell diagram shows the BP-RP color and luminosity of stars in the Gaia catalog (Copyright: ESA/Gaia/DPAC, CC BY-SA 3.0 IGO).



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.

Formatting queries

The queries we have written so far are 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 better 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'
```

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)
```

In this example, the variable that contains the column names and the variable in the format specifier have the same name. That's not required, but it is a common style.

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

```
query3
```

```
'SELECT \nTOP 10 \nsource_id, ra, dec, pmra, pmdec\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
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:

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

```
<Table length=10>
   name    dtype    unit                              description
--------- ------- -------- ------------------------------------------------------------------
source_id   int64          Unique source identifier (unique within a particular Data Release)
       ra float64      deg                                                    Right ascension
      dec float64      deg                                                        Declination
     pmra float64 mas / yr                         Proper motion in right ascension direction
    pmdec float64 mas / yr                             Proper motion in declination direction
Jobid: None
Phase: COMPLETED
Owner: None
Output file: sync_20210315091929.xml.gz
Results: None
```

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

Table length=10

source_id	ra	dec	pmra
	deg	deg	mas / yr
int64	float64	float64	float64
5895272561481374080	213.2606587905109	-56.55044401535715	0.3894438898301715
5895261875598904576	213.5508930114549	-56.61037780154348	0.16203641364393007
5895247444506644992	213.33215109206796	-56.975347759380995	-7.474003156859284
5895259470417635968	213.78815034206346	-56.64585047451594	-5.287202255231853
5895265925746051584	213.17082359534547	-56.74540885107754	-7.880749306158471
5895260913525974528	213.66936020541976	-56.66655190442016	-4.7820929042428215
5895264212062283008	213.7755742121852	-56.51570859067397	-6.657690998559842
5895253457497979136	213.30929960610283	-56.78849448744587	-5.242106718924749
4143614130253524096	269.1749117455479	-18.53415139972117	2.6164274510804826
4065443904433108992	273.26868565443743	-24.421651815402857	-1.663096652191022

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.

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`.

In the next lesson we will extend these queries to select a particular region of the sky.

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 2000 rows, you can run it synchronously, which might complete faster. If it might return more than 2000 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.

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.