PyVO

European Science Cluster of Astronomy & Particle physics ESFRI research Infrastructures

Hendrik Heinl, Markus Demleitner, Dave Morris

February 10, 2021

Abstract

The Astropy affiliated package PyVO provides APIs to many VO protocols. The flexibility of these APIs especially in regard of interoperability, big data and ML makes PyVO a powerful tool for the data scientists in astronomy. Here we give a brief introduction to the funcionality. A comprehensive PyVO introduction is linked in section References below.

Software: PyVO, TOPCAT V4.8

1 Starting out

We will use the VO tools PyVO, astropy and TOPCAT. These can be found here:

TOPCAT: `http://www.star.bris.ac.uk/~mbt/topcat/`

PyVO and astropy are best installed from your distribution packages (we do not recommend python packagers like pip, anaconda or alike).

2 PyVO Service Objects

PyVO is the Python API to many of the VO protocols. You can use it for data discovery and data access in the same way as you would use VO clients like Topcat or Aladin. But you can also use it embedded in your own code, so that you may access data remotely from an automated script.

PyVO Service Objects for the Simple Conesearch (SCP), Simple Image Access (SIA), Simple Spectra Access (SSA) and Simple Line Access (SLA) follow the general convention of:

```
pyvo.dal.SERVICETYPE(parameters)
```

Of course the parameters vary depending on the actual Service type you use: images will demand a specification of a field of view, spectra need a range of frequencies. For detailed options, please read the PyVO documentation linked in the references.

▷ **1** *Example 1: PyVO and SIA –*

Have a brief look at the few lines in example1.py. Note, that two lines are necessary to suppress warnings – services and clients often are a bit off, especially in regard of recent standards. They still work, but warnings will be raised. We don't want to pretend that everything in the VO is perfect, but within a tutorial, warnings might be confusing. If you are curious what's going on, simply comment the according lines and run the scripts with warnings enabled.

First we need to define an access url:

```
access_url = "http://dc.g-vo.org/lswscans/res/positions/siap/siap.xml?"
```

How you do find these URLs is a question of data discovery – and there is a lot to say about this, but it's a complete different topic. Just think of it as using Topcat or Aladin or any other feasable VO-tool to explore data services by selecting sources manually, and using PyVO for your aim for the mass of sources and data.

Then we build the Service Object:

```
service = pyvo.sia.SIAService(access_url)
```

If you use an interpreter like ipython3 to follow the script, you may notice a slight delay that happens when you buld the service object. Whilst building the object, general information about the services is obtained. You can find a description of the service with `service.describe()`

Eventually you see that you can query the service with the `search()` method. A few parameters need to be applied, in case of SIA these are at minimum a position in (ra, dec) and a rectangle defined by the distances in degrees to either sides from the center. `results = service.search((10.0, 35.0), (0.5, 0.5))`

Note: the search does not download any images. The results object instead contains metadata on the images that fit the criteria of the parameters. The results object is an iterable object, so one can access the metadata for each of these images and could make a selection of which images are of interest to download.

```
for image_desc in results:
   print(image_desc.title, image_desc.dateobs, image_desc.filesize)
```

With this information we can have a better understanding of which image to download, which can be done with

```
image=results[0]  image.cachedataset()
```

PyVO APIs work all similar, so it is quite easy to apply a similar selection to Cone Search Services and using PyVO Cone Search Classes.

▷ **2** *Example 2: PyVO and TAP –*

PyVO offers an API to TAP. TAP Services require a query in form of the Astronomical Data Query Language. This on the one hand sounds a bit more complicated, but the actual python code is simpler. First we build die Service Object:

```
service = pyvo.dal.TAPService ("http://dc.zah.uni-heidelberg.de/tap")
```

Then we start the query:

```
result = service.search ("SELECT TOP 1 * FROM ivoa.obscore")
```

At this point, the understanding of ADQL is not necessary. It's enough to know that the query means "Select the first row of table ivoa.obscore". The result of a TAP query is always a VOTable, so here it is a astropy votable object. Note: this are special astropy tables with a bit different methods than normal astropy tables. Because there is a great focus on interoperability, you can easily save the table into a local file:

```
result.votable.to_xml("result_example2.vot")
```

If you are familiar with ADQL, you can make much more complex (and messier looking) queries, but these give you a lot more power on a remote data selection.

▷ **3** *Example 3: PyVO, async TAP_Upload, ObsCore, Topcat and SAMP –*

For this step we need to start Topcat, because instead of saving our result to a local folder, we are going to send it to Topocat via SAMP.

We start out with reading a local table of neutrino candidates from the Antares experiment (which in fact we obtained with a Cone Search). But here we read it from the local file:

```
lt = Table.read('neutrinos.vot', format='votable')
```

Then we build the TAP Service Object:

```
service = pyvo.dal.TAPService ("http://dc.zah.uni-heidelberg.de/tap")
```

And we run the query with

```
result = service.run_async(query=query, uploads="lt":lt)
```

So first of all, we see the `.run_async` method. TAP services can be queried in synchronous mode or asynchronous mode. The first is recommended for fast ("small") queries that usually run for 2 minutes most. The asynchronous mode usually is for queries which which run for up to 2 hours (this is up to the service providers). The asynchronous mode gives you a little more control, because you usually can decide when to download a result (see ADQL and PyVO documentation linked in the References).

The `query` was define outside the main function and is a combination of TAP_Upload and positional crossmatch with a `JOIN` of the local table with the remote table. The query is askin the GAVO datacenter "Do you have image data for any of the position of my local table ?" where the position are taken from the columns RA and Decl, and the angle of the cone selection from the

Column Beta. The parameter `uploads` specifies the astropy Table Object that we need for this TAP_UPLOAD.

For dramatic reasons it's recommended to put Topcat into focus before running the script, becaus then one can see how the data is loaded into Topcat's table stack via SAMP:

```
result.broadcast_samp("topcat")
```

Hopefully we succeeded to give you a good impression of the possibilities of PyVO and how it might make you life as a datascientist easier.

3 Examples and Tables

example1.py An example program for doing a trivial SIAP query

example2.py An example program for doing a trivial TAP/ADQL query

example3.py Putting it together: a simple program to import a local table, perform a asynchronous TAP_UPLOAD on an ObScore Services and sending the result to Topcat via SAMP

neutrinos.vot Table with positions of neutrino candidates which is needed for the TAP_UPLOAD in example3.py

The examples in this presentation are copied and simplyfied from a comprehensive tutorial presented at ADASS 2020. You find the tutorial with the example scripts on github:
Hyantis on github

4 Standards

ADQL

ObsCore

SIA

SAMP

TAP

VOTable

5 References

Demleitner, M., Heinl, H. ADQL-Course

Demleitner, M., Heinl, H. PyVO-Course

Demleitner, M., Becker, S. PyVO Documentation

Taylor, M. TOPCAT Documentation