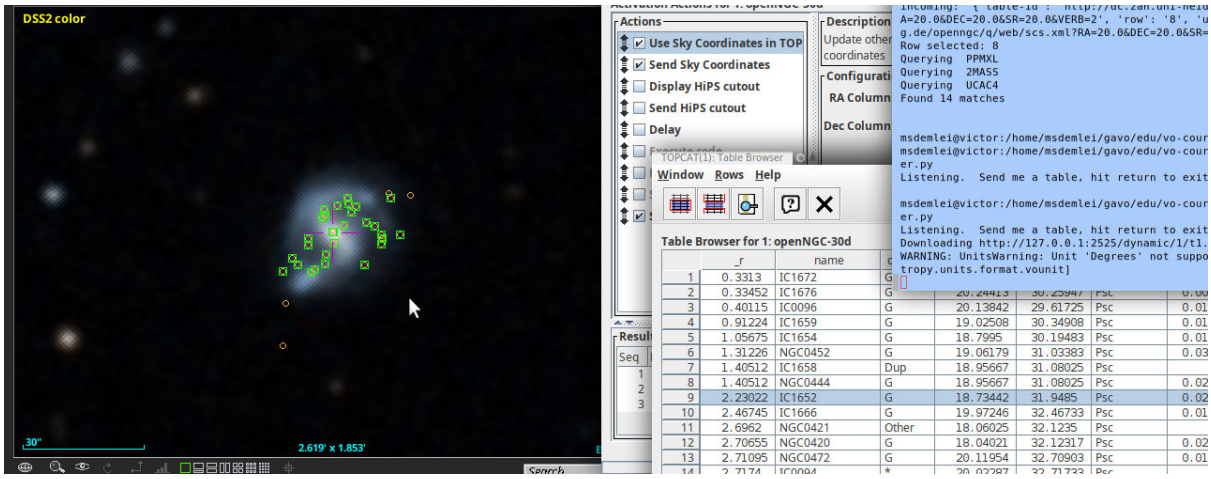
A Short Course on pyVO

Markus Demleitner Hendrik Heinl

March 20, 2025



Abstract

This is a course on pyVO, an astropy-affiliated Python library implementing client parts for many protocols in the Virtual Observatory: Simple discovery protocols like SCS, SIAP, and SSAP as well as the sophisticated Table Access Protocol TAP, which allows users to send complex queries to remote tables and retrieve metadata-rich results. There is also an interface to the VO Registry to enable data and service discovery.

The course comes with many exercises, most of which also have solutions. We hope it is suitable for both self-study and as lecture notes in teacher-led situations. Participants should have a working knowledge of Astronomy, Python and ADQL.

Contents

1 Introduction

This course will introduce you to the primary concepts of pyVO, an astropy-affiliated package for accessing Virtual Observatory services from Python. It is too much for a day in an interactive situation, so if you are reading this at the beginning of a course day: Say what you are interested in – we will have to select material anyway.

The course assumes familiarity with VO concepts (services, protocol types, the registry) as well as astropy, but you can probably gather missing parts as you go (or ask, if you are reading this in an interactive course situation). You should know enough of ADQL to be able to understand and edit queries.

The course is structured into an general introduction that you should at least cursorily read, and several more advanced topics that can be independently studied. The general introduction covers a few general patterns for pyVO usage, and it discusses the basics of the Table Access Protocol TAP, which arguably is the most versatile protocol in the VO. It does this along a few more or less contrived use cases designed to touch the central topics.

Examples for the more advanced topics in the second part include receiving SAMP messages, using EPN-TAP, or Datalink. Look at these as need arises.

The full source code for the programs discussed here is also available as an attachment if you read this in pdf. One way to retrieve them is to get pdftk (there are packages for it for Debian-derived systems), run `pdftk pyvo.pdf unpack_files`. Other PDF tools may also support attachments. For instance, in KDE's Okular its at File/Embedded Files, and in Adobe's proprietary Acrobat Reader 9, attachments can be retrieved through the paperclip icon in the lower left corner. Tools may also save attachments when you click on the Ψ icons. If all else fails, `git clone` the course's source repository (see below). All attachments are in there, too.

It is also a good idea to have a browser tab open on pyVO's documentation[1] as well as astropy's documentation[2].

The sources for everything (including these notes and the slides) are available in a git repository at codeberg.org[3]. Feel free to file bugs or even merge requests.

[1] http://pyvo.readthedocs.io/en/latest/
[2] http://docs.astropy.org/en/stable/
[3] https://codeberg.org/msdemlei/pyvo-course.git

What is the VO?

The VO is a set of standards that let clients discover and interrogate astronomical data services in a uniform manner. Standards include:

- Registry – describing and finding services
- VOTable, UCD – writing tables with rich metadata
- SAMP – connecting software components
- SCS, SIAP, SSAP – querying catalog, image, and spectral services
- TAP – running remote database queries
- Datalink – bundling up complex data and services
- MOC, HiPS – sky coverage and hierarchical imaging

The purpose of all this is so machines instead of humans can operate the services. With an average web page, that's hard to impossible.

Machines operating services, in turn, are important to save manual work. This is part convenience, but mainly it is so you can use more and diverse data for your research.

See the IVOA home page[4] for more information.

We will speak almost all of the protocols mentioned above within this course, but there is no need to dig into what all of them do here – they will come in quite naturally when we want to solve problems.

2 pyVO Basics

Prerequisites

- python and astropy, of course (we assume Debian stable, at least; anaconda on proprietary systems should do, too)
- TOPCAT[5] for viewing and visualising tables
- Aladin[6] to work with images
- pyVO. Get it from
  - https://pypi.python.org/pypi/pyvo
  - or try `apt-get install python3-pyvo`
  - or try `pip install pyvo`
  - or try `conda install pyvo`

[4] http://ivoa.net
[5] http://www.star.bris.ac.uk/~mbt/topcat/
[6] http://aladin.u-strasbg.fr/aladin.gml

Python Matters

In this course, we will use python scripts most of the time rather than the jupyter notebooks you may be more familiar with.

This is partly personal preference, but for "production" scripts have several important advantages:

- Meaningful version control
- Can use proper editors
- Files can work as modules

However, if you prefer notebooks, you can use pyVO from Python notebooks, too. If you are unsure how this looks like, see the attached tap-obscore.ipynb (which covers several of the topics we will later discuss).

Ψ tap-obscore.ipynb

To fit things on slides, I am PEP 8-relaxed. PEP 8[7] is a set of relatively sensible rules for how you should format your Python source code so other people want to read it. I am not always following it here. In particular, on slides, I am using indents of two spaces against the PEP 8 standard of four, which you may need to fix when cutting and pasting.

What's pyVO?

pyVO provides APIs for lots of VO protocols.

It is glue between astropy and python in general and the astronomical data services in the VO.

It is a community project. You are most welcome to contribute at
https://github.com/astropy/pyvo.

Running Simple Services

When querying "simple" remote services (image, spectral, cone search; *not* directly TAP), pyVO has a consistent pattern:

```
# <prot> is SIA, SSA, SCS, SLA...
import pyvo

# construct a service object with a service's endpoint URL
service = pyvo.dal.<prot>Service(access_url)

#call the search method with the protocol's parameters
for result in service.search(<parameters>):
  ...work on dict-like object result...
```

The "dal" in here means "Data Access Layer", which essentially means: the VO protocols dealing with how to query services and how the services are supposed to respond.

You will soon learn how to find out the access URLs.

[7] https://peps.python.org/pep-0008/

Query a Single Image Service

Example: SIAP, the VO's protocol to access image servers.

Query a VO service for a list of images covering a small field on the sky, and download one of these images:

```
svc = pyvo.sia.SIAService(ACCESS_URL)
images = svc.search((340.1,3.36), size=(0.1, 0.1))
image=images[0]
image.cachedataset()
```

Ψ basicsiap.py

For SIAP, `pos` (as a tuple of ra and dec) and `size` (in degrees, either one radius or extent in ra and dec) are mandatory. More parameters: in the pyvo docs[8].

Also: `row.cachedataset` saves the image to your local disk under a name sensible for the metadata. In case the filename produced by cachedataset has an extension `.None` on your machine: that's a bug in pyVO that was fixed in 2024.

Note how you do not have to know anything about the service except its access URL. Since pyVO uses a standard protocol, it knows enough to be able to, in this case, retrieve the file and (mostly) give it a reasonable name.

This is a very basic example, though. pyVO provides you with more functionality that helps analysing the results before selecting the images. We will see some of these functions by using pyVO in a more interactive setting (e.g. ipython).

Getting source code: In case your PDF viewer gives you a hard time saving the attached Python code or does not support attachments at all, you can find all the files in this course's repository. Just run

```
git clone https://codeberg.org/msdemlei/pyvo-course
```

> **Exercise 1**
>
> Get our example basicsiap.py from the notes.
>
> Now find an image service publishing the ROSAT survey and pointed observations and see if it has an image for the position given (or try some other service and position you are actually interested in).
>
> Use WIRR[9] to search the VO Registry for now.
>
> What is coming back from SIAService's `search` is a sequence of SIARecords. Have a quick look at its pyvo documentation[10] and make your program print the file size and the instrument name rather than calling `cachedataset`.

[8] http://pyvo.readthedocs.io/en/latest/api/pyvo.dal.SIAService.html#pyvo.dal.SIAService.search
[9] http://dc.g-vo.org/WIRR
[10] https://pyvo.readthedocs.io/en/latest/api/pyvo.dal.SIARecord.html#pyvo.dal.SIARecord

This is Python

The advantage of doing this in Python is that it is easy to add your own logic. Here is how to add time constraints (SIAP version 1 unfortunately does not specify how to tell the service you are only interested in a specific time interval – we will later see how more modern standards let you push time constraints to the server) and search multiple positions:

```
svc = pyvo.sia.SIAService(ACCESS_URL)
for pos in [
    (213.97, 11.50),
    (230.44, 52.92)]:
  images = svc.search(pos, size=(0.5, 0.5))
  for row in images:
    if not DATE_MIN<row.dateobs<DATE_MAX:
      continue
    row.cachedataset()
```

Ψ multisiap.py

A word on `row.dateobs`: While SIAP (as most of the VO) delivers dates as modified julian dates (MJD), pyVO turns these values into `astropy.time.Time` instances. You could turn these back into floats (my taking their `.mjd.real` attribute) and compute with MJD yourself, but it is smarter to keep your times in `Time` instances, too, as shown in the multisiap.py.

Metadata in pyVO

You can access the metadata coming with the response VOTables from pyVO, too, albeit somewhat obscurely:

```
>>> import pprint
>>> pprint.pprint(images.votable.infos)
[<INFO ID="legal" name="legal" value="The data from Maydanak observatory
>>> pprint(images.votable.resources[0].infos)
[<INFO ID="queryPars" name="queryPars" value="(%(siaarea0)s &amp;amp;amp; c
 <INFO ID="QUERY_STATUS" name="QUERY_STATUS" value="OK"/>,
 <INFO ID="request" name="request" value="/maidanak/res/rawframes/siap/s
 <INFO ID="standardID" name="standardID" value="ivo://ivoa.net/std/sia"/
 <INFO ID="server_software" name="server_software" value="DaCHS/2.9.3 tw
 <INFO ID="server" name="server" value="http://dc.zah.uni-heidelberg.de"
 <INFO ID="citation" name="citation" ucd="" value="http://dc.zah.uni-hei
 <INFO ID="citation" name="citation" ucd="" value="http://dc.zah.uni-hei
 <INFO ID="ivoid" name="ivoid" ucd="meta.ref.ivoid" value="ivo://org.gav
```

For why the information is available in this way, you need to understand a bit of VOTable. But this pattern works for all responses you will deal with in current VOTable.

Excursion: The Python Debugger

To inspect metadata like this from within a running program (as opposed to a notebook), it is really convenient to use the python debugger. To drop into it, call `pdb.set_trace()`:

```
for pos in [
    (150.36, 55.90)]:
  images = svc.search(pos, size=(0.5, 0.5), verbosity=2)
  import pdb;pdb.set_trace()
  for row in images:
```

You can then enter Python statements (like the info expressions) and do many other things described in the Python reference[11]. When done looking around, you can type `cont` to let your program continue of `quit` to exit it.

And now all-VO

The nice thing about standard services: Handle one, and you get them all. So, let's add a query to the Registry and run our query all over the VO –

```
for svc in registry.search(servicetype="sia", waveband="optical"):
  try:
    search_one_service(svc.accessurl)
  except Exception:
    import traceback; traceback.print_exc()
```

Ψ globalsiap.py

Wisdom: In multi-service queries, expect at least one service to be broken. Write your scripts to cope.

The `registry.search` function we are using here interfaces to a big directory of all the services that are in the VO: The Registry, which is also what is underlying the WIRR web page employed in problem 1.

The way we are querying the Registry here is a bit simplistic. In particular, you probably do not want to use *servicetype* constraints when doing science work. Global dataset discovery (which is what we are approaching here) is a lot more involved than just querying all services of a type (although this used to somewhat work in the early days of the VO). For now, however, we when we query like this, for everything that comes back from `registry.search`, we can request an image ("SIA") service. This happens in `search_one_resource` with

```
svc = res_rec.get_service("sia", lax=True)
```

Accept the `lax` argument for now. We will have a closer look at pyVO's Registry API later.

The exception catcher is there since not all services claiming to be standards-compliant actually are. It does not hurt to complain to the service operators if a service you are interested in behaves weirdly – sometimes the operators simply have not noticed that it is broken, or possibly has just broken.

To find out who to complain to, you can again use the Registry; the objects that are returned from `registry.search` have a `get_contact` method:

```
>>> svcs = pyvo.registry.search(keywords="pyvo")
>>> svcs[0].get_contact()
'Hendrik Heinl (+49 6221 541849) <gavo@ari.uni-heidelberg.de>'
```

You will probably also see lots of warnings from astropy's VOTable parser. This is partly because astropy is overly paranoid, rejecting UCDs actually required by the SIAP standard, partly because operators botch things. Interoperability is not always easy. At this point it is probably too early to complain to operators about astropy's VOTable warnings. We will later turn them off.

If a service hangs, you can interrupt it by hitting Control-C. In production code, you can set timeouts. We will later see how to do that.

[11] https://docs.python.org/3/library/pdb.html

Exercise 2

Get the globalsiap.py script from the attachment and change it so it skips 90% of the services discovered randomly (use `random.random()`). Also, remove the constraint on the date (we don't need that here) and change the position to something you are interested in or expect to have pretty pictures (M1 or M51 are always good candiates). Run the thing and see what you find.

Add SAMP Magic

SAMP lets you exchange data between VO clients. Your script is a VO client, too. Let's make it broadcast some of the found images:

```
with pyvo.samp.connection() as conn:
  ... (search) ...
  pyvo.samp.send_image_to(conn, image.acref)
```

Ψ globalsiapsamp.py

Before running this, start Aladin (or some other SAMP-enabled image client) so the images are displayed.

In general, SAMP-enabling programs may not come quite natural to people who so far have mainly written fairly linear science code, because when doing SAMP you usually want to react to external events. In linear code this is rather uncommon.

In this example we are just sending data, which does not require much reacting to external signals. We still have to manage the connection to the SAMP hub – things get ugly if you do not properly close the connection –, which is taken care of by a context manager from `pyvo.samp`.

A context manager is a python construct consisting of an opening line of the form `with cm [as name]:` and then a block, the "controlled block". It is designed to ensure what is called "external invariants", some piece of state that the system should be in outside of the controlled block. You may know this from files, where the external invariant is "the file is closed":

```
with open("test.txt", "w") as f:
  f.write("some content\n")
print("f is closed")
```

By the time the print statement is reached, Python's semantics guarantee that f is closed and the content is written, regardless of what else happened (think exceptions) happened in the controlled block. The SAMP connection similarly ensures that once the controlled block is left, the connection is closed.

Given we are doing function calls between different processes written in different languages, we would argue this kind of code actually is surprisingly compact.

Exercise 3

Get the pyVO source code and find the source of pyvo.samp. Start TOPCAT, find the implementation of the `connection` context manager, and then open a SAMP connection manually from an interactive Python prompt. And then again, and a third time. What do you observe in TOPCAT?

Hint: To get the source code, try:
`git clone https://github.com/astropy/pyvo`.

Or, on Debian-derviced boxes:

```
apt source python3-pyvo
```

Exercise 4

Still in `samp.py`, inspect how `send_image_to` is implemented. From reading the code, can you figure out how to only send the image to Aladin? If you can, try your solution in globalsiapsamp.py by having Aladin and ds9 (Debian package: saods9) open at the same time.

Hint: To find out Aladin's client name, check TOPCAT's SAMP status window.

3 pyVO and TAP

Enter TAP

What we have seen so far does not scale when you are interested in more regions.

Also, only fairly basic constraints are supported.

TAP is far more powerful.

Sample use case: Integrate photometry from different source catalogues, do some local work on results, try to obtain spectra for interesting candidates.

Run Sync TAP Queries

Run queries via TAP:

```
access_url = "http://dc.g-vo.org/tap"

service = pyvo.dal.TAPService(access_url)
result = service.run_sync(
  """SELECT raj2000, dej2000, jmag, hmag, kmag
        FROM twomass.data
        WHERE jmag<3""")
for row in result:
  print(row["raj2000"], row["jmag"])
```

Exercise 5

Write a program that prints the number of rows in the table `arihip.main` in the TAP service at http://dc.g-vo.org/tap (do *not* pull all the rows and use python's len).

Hint: With ADQL's `AS` construct you can control the names of table columns.

This is another instance of the pyVO pattern "create a service object, then call a method". In this case, we are calling `run_sync` – this is not called `query` as for the other services because TAP has two modes of operation; we will get to the other one (unsurprisingly called async) in a moment.

What is coming back from `run_sync` is a sequence of `dal.Record` elements (well, the truth about TAPResults[12] is a bit more complex, but that's the gist of it).

You can make a normal astropy table from the result by calling `result.to_table()`, and there often are good reasons to do that. For instance, to save the table to a disk file, you can write:

```
result.to_table().write("saved.vot", format="votable")
```

[12] http://pyvo.readthedocs.io/en/latest/api/pyvo.dal.TAPResults.html

Step 1a: Multiple TAP Queries

```
# Imagine more interesting queries here.
QUERIES = [
  ("twomass", "http://dc.zah.uni-heidelberg.de/tap",
    """SELECT TOP 1000000 raj2000, dej2000, jmag, hmag, kmag
      ...CIRCLE('ICRS', {ra}, {dec}, {radius}))"""),
    ...}

with pyvo.samp.connection() as conn:
  for short_name, access_url, query in QUERIES:
    service = pyvo.dal.TAPService(access_url)
    result = service.run_sync(query.format(**locals()), maxrec=90000)
    pyvo.samp.send_table_to(
      conn,
      result.to_table(),
      client_name="topcat",
      name=short_name)
```

Ψ fetch3.py

This does several things we have not seen before:

- `QUERIES` is a sequence of tuples; for examples, check the full source. Tuples are often a good choice when you have "inhomogeneous" (e.g., each item in a sequence "means" something different) data without much behaviour. When the rows become more complex, consider using python's `dataclasses` module, and when they have non-trivial behaviour, a "normal" class. Here, we just group a service title, a service URL, and a template for the query to run, for which a tuple works nicely.

- `query.format(**locals())` is a trivial example of what's called *templating*; you write a string that gets filled in, in this case using python's plain `format` method. You can (and sometimes should) get a lot more fancy with templating; one reason to do that could be to automatically quote strings. But as long as you control both the template and the fillers, it is probably better to not pull in extra dependencies just for templating.

  `**locals()` is a way to say: make all local variables available as keyword arguments. In general, `**` in an argument list means: what's next is a mapping, and turn it into keyword arguments, which sometimes is convenient if you want to build up a set of arguments step by step.

- `maxrec=90000` asks the server to return up to 90'000 rows (the *match limit*). When you do not pass *maxrec*, a service-specific default kicks in; you can find that default at `service.maxrec` (but take it with a grain of salt; this *may* be something like a lower limit). PyVO will issue a warning if your result overflowed your maxrec.

- `pyvo.samp.send_table_to` does a SAMP transfer of an astropy table (hence the `.to_table()`) to a SAMP client; it does a broadcast if you do not pass a `client_name`.

Exercise 6

The following program should print URIs and titles for images in some collection for whatever names are in `OBJECTS`:

```
import pyvo

OBJECTS = ["IC 4756", "NGC 3377"]
```

```
QUERY = """select accref, imagetitle
    from maidanak.reduced
    where object={object}"""

svc = pyvo.dal.TAPService("https://dc.g-vo.org/tap")
for object in OBJECTS:
    print(svc.run_sync(QUERY.format(**locals())).to_table())
```

(Note: this is *not* the way to match against multiple objects; you would instead use SQL sets or, probably more commonly, TAP uploads outside of silly exercises).

What really happens: An error message. Can you figure out where it comes from and how to fix things?

Exercise 7

Use TOPCAT's TAP data browser to locate services and table names for TGAS and RAVE (or just use the GAVO DC TAP service with tables tgas.main and rave.main). Also figure out where the positions and some usable magnitude are, plus the proper motions from TGAS and the radial velocities from RAVE (or just blindly use ra, dec, pmra, pmdec, phot_g_mean_mag for TGAS and raj2000, dej2000, rv, and hmag for RAVE).

Re-write fetch3.py to query the retrieve all stars between 8 and 8.2 mags from each table (don't worry about the difference between H and G magnitudes for this problem). Also, send the results to Aladin (which is known as *Aladin* (capitalised) on the SAMP bus). See if you can get a nice plot of rv, pmra, and pmdec.

Hint: Check Aladin's Catalog/Create filter for fancy plotting options.

Step 2: Go Async

When doing a lot of queries or long-running queries, run them asynchronously and in parallel.

Asynchronous means that you go to a service, submit your query there and immediately receive some sort of token. With this token, you can come back later and retrieve your result. In the meantime, you are free to do whatever else you have to do – which includes turning off and/or moving your machine, for instance.

In this case, the main advantage is that we can run our queries in parallel. If all you want is have more time for your query, see the next slide for simpler options to run async TAP jobs.

```
jobs = set()
for short_name, access_url, query in QUERIES:
  job = pyvo.dal.TAPService(access_url).submit_job(
    query.format(**locals()), maxrec=9000000)
  job.run()
  jobs.add((short_name, job))

 while jobs:
  time.sleep(5)
  for short_name, job in list(jobs):
    if job.phase not in ('QUEUED', 'EXECUTING'):
      jobs.remove((short_name, job))
      pyvo.samp.send_table_to(...)
      job.delete()
```

Ψ fetch3-async.py

We told you sync is easier to program with. But on the other hand: With this program, all three queries run in parallel, which is nice, in particular if they take a while. Additionally, you have a little more control about when to receive the data.

What's happening here? First, we submit all jobs. Rather than `run_sync` we now use TAPService's `submit_job` method. While taking the same arguments as `run_sync`, it immediately returns. Since it cannot peek into the future, it cannot return the finished result. Instead, you get an object that one can use to manipulate the remote job. That remote job is *not* started by `submit_job`. It is instead waiting for further configuration (e.g., increasing its maximal runtime) or a request to put it into the processing queue.

For our task, it is enough to just start the job using the `run` method. We then add it to a watch set of running jobs.

The rest of the code above is all about managing this set. In a polling loop – be sure to introduce sleeps or your code will hit the remote services all the time – we iterate through the jobs. Actually, we iterate over a copy of the job set since we want to delete completed from it, and we couldn't do that if there was an iterator over it active.

In the loop body, we check the `phase` attribute of the job. Although this looks like an attribute access, in each iteration pyVO goes to the remote service and asks it what our job is doing. While it is in either QUEUING or EXECUTING states, it is still worth waiting for a result. Other states include PENDING (not yet started), COMPLETED (done, result available), ERROR (done, some kind of failure happened; call the `raise_if_error` method to turn it into a python exception), and ABORTED (interrupted by client or operator intervention).

Once we find a job is done, we remove it from the job list and send the result over to TOPCAT as before.

Finally, we delete the remote job. That's a nice thing to do. Services will eventually delete your job anyway (you can figure out when and even change that date in the job's `destruction` attribute), but it is good style to discard jobs once you do not need them any more.

This example is primarily intended to illustrate async mode itself.

Lightweight async

If you can live without real-time monitoring, you can write more concisely:

```
job.wait()
job.raise_if_error()
result = job.fetch_result()
```

In its default configuration, `job.wait()` waits for a change in the job status or a timeout and then returns. On modern TAP services, this generally is only one request every 10 minutes or so; this saves server-side ressources.

The `raise_if_error()` method gives you more reasonable exceptions than if you blindly try to access results from jobs that failed server-side.

With only a single job at a time, it is even simpler:

```
result = svc.run_async(query, ...)
```

The interface of `run_async` is that of `run_sync`, i.e., it will block until the results are in. Use it if you have to go async because your job runs too long for sync (in general, sync jobs have to finish in seconds to minutes, while async jobs can run for hours) but you want to avoid the dance with checking the phases.

Step 3a: UCDs build SEDs

Can we build SEDs from the results of the three services?

Not simply; photometry metadata in the VO is not quite sufficient for that yet. However, UCDs let us do a workaround:

```
UCD_TO_WL = {
  "phot.mag;em.opt.u": 3.5e-7,
  "phot.mag;em.opt.b": 4.5e-7,
  "phot.mag;em.opt.v": 5.5e-7,
  "phot.mag;em.opt.r": 6.75e-7, ...}

  for row in rows:
    for index, col in enumerate(row):
      ucd = row.columns[index].meta.get("ucd", "").lower())
      if ucd.startswith("phot.mag"):
        if ucd in UCD_TO_WL:
          phots.append((UCD_TO_WL[ucd], col))
```

Calling our multi-band data a SED ("Spectral Energy Distribution", that is some sort of flux densities plotted as a function of the spectral coordinate) is perhaps somewhat pretentious. To make this an actual SED, we would at least have to worry about photometry systems, which is a *real* concern even in the narrower optical, not to mention when you leave the optical. But bear with us.

UCDs ("Unified Content Descriptors") are VO-standardised strings defining the physics contained in columns. They even have a bit of syntax. In our example, we can see that first, we have magnitudes ("phot.mag") and then that they were taken in a certain band.

Similarly, "pos.eq.ra" would tell you that something is a right ascension as part of an equatorial position; since tables sometimes have multiple positions in a single row (e.g., different reduction, position in some reference catalogue, or position of a sub-feature), you may want to single out a particular column as your preferred, primary, default, or whatever RA. For that, use "pos.eq.ra;meta.main".

UCDs are particularly nifty in data discovery when you are looking for tables that have a certain kind of physics. Of course, that only works when people properly mark up their tables with UCDs – be sure to do that on your data whenever you let a VOTable leave your disk. The full list of UCD atoms is available from the IVOA document respository[13].

The clean way, incidentally, is a proper annotation of the columns in question with full photometry metadata (e.g., central wavelength, bandwidth, the system, perhaps a URL of the detector's response curve, etc). The details are hellish, but there actually is a photometry DM in the VO. There is just not a good way to put that information into a VOTable yet. If you are looking for something to contribute to the VO: this would be a good task. Just ask on the IVOA's data models mailing list.

Step 3b: Aggregate Photometry

Construction of "clusters" is in vohelper.py and uses astropy's `SkyCoords` and `match_catalog_to_sky` (asymmetric!).

For three catalogues, we must perform six sky matches to get pairs, then walk the graph to gather the clusters.

[13] http://www.ivoa.net/documents/latest/UCDlist.html

This actually is pure astropy and has nothing to do with pyVO as such. As a matter of fact, it is usually smarter to have the remote sides do the cross matches if at all possible.

In this case, since we do not have a "master catalogue" to match against, that is actually hard. For smallish crossmatches, the code in vohelper works reasonably well (but it scales horribly when then number of tables increases; use specialised packages when your problem takes that direction).

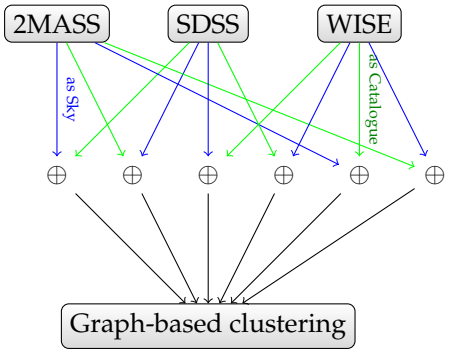
What is happening in that code? `sky_coords` are `astropy.SkyCoord` instances (in the example code, there is a function `get_coordinates_for_table` that makes these for essentially arbitrary tables as long as they are properly marked up).

The code then goes through all pairs of input `SkyCoords` and uses their catalogue match method to generate pairs of indices into these objects that are the closest pairs (that operation is not symmetrical, which is why we compute the matches with all permutations).

The remaining code filters out those pairs that are closer than a limit that is passed in and adds a new pair of rows to be matched to a set. Each row is designated as a pair of table index and row index within that table.

The rest is a graph problem: If you compute the connected subsets of the graph formed in this way, you will have all measurements that are crossmatched together and thus, hopefully, correspond to one object.

Sorry for this excursion. Feel free to ignore this.



For this course, but perhaps also for convenience in wider usage, we have gathered some helper functions in a module vohelper that you can find on the web page and attached to the PDF. Have a glance at the source code if you want. Otherwise, just dump it next to your scripts so you can import it.

Ψ vohelper.py

Combine with "your" Code

This is python: Add your own logic!

Here: Let's display the approximate SEDs and let the user interactively select "interesting" cases.

```
for pos, phots in seds:
   to_plot = np.array(phots)
   plt.semilogx(to_plot[:,0], to_plot[:,1], '-')
   plt.show(block=False)
   selection = input(
     "s)elect SED, q)uit, enter for next? ")
   if selection=="q":
     break
   if selection=="s":
```

```
      selected.append(pos)
    plt.cla()
  return selected
```

Ψ fetch3-cluster.py

This is fairly standard matplotlib. We are interacting through `input` in the shell here for simplicity. It is not actually hard to interact through the matlotlib window, but that requires a bit object magic that we wanted to avoid here.

Exercise 8

Go through the source code of fetch3-cluster.py. You will see we have put in two workarounds for where the data providers messed up. Can you see in each case what might have gone wrong? Have the service operators fixed their software or do things still fail when you remove a workaround? In a course setting, coordinate with your neighbours and split up the work so each only looks at one workaround.

Exercise 9

Run fetch3-cluster.py and select a couple of objects. Keep the resulting file (`selected_positions.vot`) – we will want to reuse it later.

Write Tables in Style

Please furnish your tables with metadata. fetch3-cluster shows you how to do it with astropy:

```
t = table.Table()
t.add_column(table.Column(
  name='ra',
  data=selected[:, 0],
  unit=u.degree,
  description="ICRS RA of a selected object",
  meta={"ucd": "pos.eq.ra;meta.main"}))
```

Looking for Spectra

Suppose you have a couple of positions for "interesting" objects. Can we find spectra for them?

SSAP is the traditional VO protocol to access spectra, quite like SIAP, and we could query SSAP services just like we queried SIAP services. However, SSAP only lets you access one object at a time, which is kind of tedious.

Let's use

ObsTAP = TAP with table `ivoa.obscore`

`ivoa.obscore` has lots of metadata on observational data products (spectra, cubes, timeseries).

Having what people generally call a "data model" – here, rather a set of pre-defined columns – enables a lot of powerful data discovery scenarious when coupled with TAP. So, why do we bother with SCS, SIAP, and SSAP?

Good question. It mainly has historical reasons – the S-protocols where easier to define than TAP and Obscore. And until datalink was there, there were a few tricks you could play with them that just do not work with simple ObsTAP (cutouts, for instance).

Even now, there is still much less data in ObsCore services than in SSAP; hence, if your problem easily admits querying through SSAP, it is certainly no mistake to do so, perhaps in addition to Obscore (beware: there is some data that's in Obscore but not in SSAP).

What we are doing here is another instance of the more general problem of global dataset discovery, to which I will return later in more generality.

Plan:

- Search for ObsTAP services
- Use TAP upload to search to collect spectra
- Send spectra to SPLAT

Obscore

The obscore "data model" consists of $\sim$ 40 columns; use a TAP browser to look at them. Some highlights:

- dataproduct_type – states *image*, *timeseries*, and the like. The full list of terms is at http://www.ivoa.net/rdf/product-type.
- obs_publisher_did – a dataset identifier. By design, it should be globally unique and resolvable, but not all data providers are following this design…
- access_url – where to get the data from.
- s_ra, s_dec, s_fov – centre and FoV of the observation
- s_region – area covered by the dataset as an ADQL geometry. This column allows very concise queries, but alas, operators are free to have this NULL even when the have centre coordinates and a field of view.

Query the Registry

Iterate over all obscore services (here: see what data collections they house):

```
for svc_rec in pyvo.registry.search(datamodel="obscore"):
  print(f">>>>>> {svc_rec.short_name}...")
  try:
    svc = svc_rec.get_service("tap", lax=True)
    result = svc.run_sync("SELECT DISTINCT obs_collection"
      " FROM ivoa.obscore")
  except (Exception, KeyboardInterrupt):
    import traceback; traceback.print_exc()
    continue
  print("\n".join(r["obs_collection"] for r in result))
```

Do not run this script *just* for fun. It will hit quite a few services and make them seqscan their obscore tables.

To "use ObsTAP", just query the `ivoa.obscore` table via TAP.

To find TAP services having these tables, we once more use `pyvo.registry.search` but this time use the `datamodel` constraint. Also, we again use the `get_service` method on the RegistryResource instance that comes back from search; you should always specify what sort of service you want – `"tap"` in this case. Prefer this pattern over the explicit use of `access_url` on RegistryResource-s you may see in other places; access URLs are not a terribly well-defined concept, in particular not if one does not constrain the servicetype.

The selling point here is: we are running *the same* database query on all the ObsTAP services, and we are processing their results in the same way. That is the power of uniform data models.

This script does not come attached. That's because on large services, the `SELECT DISTINCT` can actually be computationally expensive for the remote side; it is likely that you will see timeouts or very long runtimes. Hence, to try it, you will have to cut and paste, and then add the pyvo import.

More useful Obscore queries with positional constraints are usually much faster: the wonder of indexes and one of the major reasons why "just download stuff" is not a good plan with large datasets.

Query with Upload

For each ObsTAP service, we query against our object list (assumed to be in an astropy Table in `pois`):

```
if not svc.upload_methods:
  return

result = svc.run_sync(
  """SELECT TOP 2000 oc.obs_publisher_did, oc.access_url
      FROM ivoa.obscore AS oc
      JOIN TAP_UPLOAD.pois AS mine
      ON 1=CONTAINS(
        POINT('ICRS', oc.s_ra, oc.s_dec),
        CIRCLE('ICRS', mine.ra, mine.dec, 0.01))
      WHERE oc.dataproduct_type='spectrum'
      """),
      uploads = {"pois": pois})
```

What is going on here? Right after constructing the service, we check whether it supports table uploads – not all TAP services do. TAPService objects have a few other attributes that let you inspect various properties of services. This, in particular, includes resource limits (maximum upload size, limit to which maxrec can be raised, etc).

Here, it is enough to know there is any upload method at all, because the standard says that inline upload must be supported if there is any upload support, and inline uploads is what we are doing.

To actually perform the upload, pass a dictionary to the `uploads` keyword argument of `run_sync` and friends. The keys there are simple names (starting with a letter and letters or numbers after that), the values can be various things, but you will probably get by passing either a string (which is interpreted as a URL to fetch a VOTable from) or an astropy table.

You can upload multiple tables using different keys; for each key, a table `TAP_UPLOAD.`*key* becomes available – this is where the `TAP_UPLOAD.pois` above comes from. Remember that TOPCAT, which is what many upload examples are written for, has the convention of naming its uploads `t<n>`, where the *n* is the index in the table list in TOPCAT's main window.

You will almost always join the uploaded table with a table on the service, and thus it is almost always a good idea to use ADQL's `AS` construct to give abbreviated names to tables. The name `mine` is typically a good choice if you only have one upload, for the simple reason that other people use it, too.

Note that even if you do not absolutely *have* to qualify column names in a query with a join with the source table names as long as the name only exists in one or the joined tables, you will regret not doing so in queries you will likely reuse – just because there's no column `s_ra` in the

table uploaded here doesn't mean the table you have in your next program does not either. If it has and you have not used the `oc.` prefix (here), your query will fail.

Instead of the common `run_sync`, this uses `vohelper.run_sync_resilient`, which catches all kinds of exceptions and other trouble. As said above, when you do all-VO queries, expect at least one service to fail completely and another to give results that look like they come from a fuzzer.

The actual obscore query does a classical, ADQL 2.0 crossmatch, because we are querying lots of services, many of which will not be updated to more recent standards even by the time your read this. Also, stellar spectra come from essentially point-like objects, and thus you probably do *not* want to write something like

```
1=CONTAINS(POINT(mine.ra, mine.dec), s_region)
```

This *could* be more attractive if you are looking for images or other artefacts with a reasonable coverage. Note, however, that proper `s_region` support is not mandatory, whereas all data providers get the center RA and Dec for their datasets roughly right. The bottom line is: If you can get by with just positions (rather than `s_region`) in your obscore queries, do it.

The code in get_spectra.py is actually a bit more general in that it does not hardcode the column names in the uploaded table but instead discovers them using UCDs. So, as long as your tables are properly annotated, the function there will just work for global spectra discovery (or, if you change the query, really any other global Obscore discovery on sets of positions).

> **Exercise 10**
>
> One particularly cool part about async is that you can keep your results publicly available on the remote server for a while. That, in turn, you can use to do cross-service joins *without having to download intermediate tables*.
>
> You can use URLs in a query's upload argument. To try this out, review the TGAS and RAVE exercise 7. Let the initial RAVE query be asynchronous. On the resulting job, call `wait` as above. Once it is done, upload what is job's `result_uri` attribute into the TGAS server with a normal positional upload join.

Collect Spectra finished

The rest is almost standard SAMP fare to get the spectra retrieved to SPLAT as they come in:

```
for ds_name, access_url in specs:
  print("Opening ...".format(access_url))
  try:
    pyvo.samp.send_spectrum_to(
      conn, access_url, client_name="splat", name=ds_name)
  except KeyError as exc:
    # regrettably, astropy raises the unspecific KeyError
    # when there it does not find the client.
    print("  ** Failed: is splat running?")
  except Exception:
    print("  *** Unexpected failure:")
    import traceback; traceback.print_exc()
```

Ψ get-spectra.py

As for images, spectra are usually passed around by their URLs in SAMP.

What is new here is that we are catching exceptions. Somewhat suboptimally (because it is too non-specific), pyVO raises a KeyError when it cannot find SPLAT on the SAMP bus.

Giving some reminder-type message probably helpful when you run the program after a couple of months and have forgotten about SPLAT being a part of this analysis chain. Letting through the KeyError with a key of *splat* is probably a lot less helpful than the message we emit, even at the risk of catching KeyErrors of different origin. In practice, you would probably want to break out of the loop, too; the way this is written, you will get one message per spectrum, which may be slightly panic-inducing.

We catch all other exceptions; we do not want to exit the loop just because some spectrum is funny. Given what is in the try-block, the most likely origin of these exceptions is when SPLAT fails to open a spectrum for some reason and sends back an indication of that. What we are catching here, in effect, are an exceptions raised within SPLAT.

In general, there is no telling if the target client has already informed the user that something is wrong – it is probably better to assume it has in generic code most of the time, and so sending code should avoid modal error messages ("Click here to continue"). But you basically never want to silence all exceptions, because that will hide all kinds of unexpected misbehaviour. So, as a relatively safe and diagnosable fallback, we just dump the traceback and trudge on.

Exercise 11

Can you change `get_spectra.py` such that only spectra of resolving power 10000 or greater are retrieved?

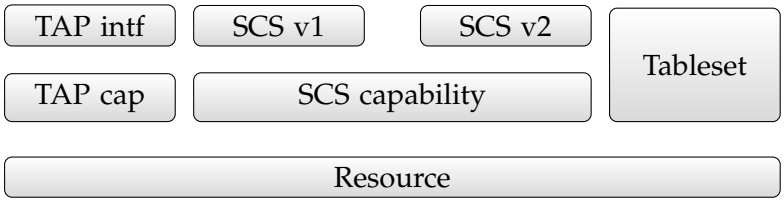
Hint: Use TOPCAT or the `tables` property of your `TAPService` to inspect the metadata of the `ivoa.obscore` table to figure out which column to query against. Just in case: It is almost always better to filter on the remote side rather than the local side. And chuck the "almost" if the constraint can be expressed as a single condition in a WHERE clause.

4 pyVO and the Registry

A Closer Look at registry.search

We have seen `registry.search` already in some places.

To go more deeply, you need to understand a bit more of the Registry data model:



The illustration shows a resource, the thing that has common metadata like a title, description, authors, space-time coverage, and the like.

On top of that sit capabilities, which are things the resource "can do for you": typically, protocol endpoints. This particular resource has two capabilities: TAP for database queries and SCS for simple cone searches.

Each capability can have one or more interfaces, that is, things that clients can talk to. For reasons of practicality, a "good" capability should only have one interface; but this may change as future standards are defined. Interfaces for multiple versions of a protocol on one capability, as sketeched here, is not something we are planning for, though; SIA1 and SIA2, the only example where that would matter right now, are modelled as two different capabilities.

There are many other things that a resource can harbour beyond capabilities; an important example is the tableset, which lists what tables the resource contains. Be warned that VODataService (the standard that defines how tablesets are written) does not *require* tablesets, and so some data publishers still do not provide them. If you catch one of those, complain to them.

Principles of RegistryResource

What you get back from `registry.search` is a sequence of `RegistryResource` instances.

It has attributes for metadata (`res_title`, `res_description`...), and important methods:

- `describe()` – return a summary of what pyVO knows about the resource.
- `access_modes()` – short identifiers for the capabilities of the resource
- `get_service(type, lax, keyword)` – return a service object to query the resource
- `get_tables()` – return a sequence of table-like objects with what tables you can query

The main method for practical use really is `get_service`. Its `type` argument is something like `"tap"` – the strings that will produces something for a given resource can be obtained using `access_modes()`.

The `lax` keyword argument deserves some explanation: If there are multiple capabilities of a given type on a resource – something that is still common for VizieR, who like to keep all tables belonging to one paper together in one VO resource in this way –, pyVO does not know which one to pick unless you pass keywords (to be matched within the capabilities' descriptions). If you think you know what you are doing, you can ask pyVO to pick one of the capabilities more or less at random: That is what `lax=True` does. It is not recommended to do that in code that matters.

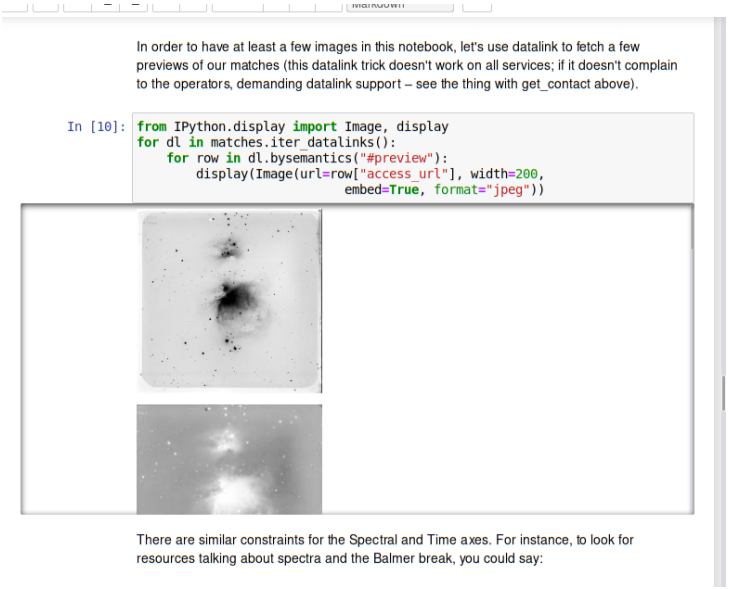
As of 2024, most of this code has recently been refurbished, and there have been bugs off and on. If you find you need to use `lax=True` when you do not expect to, it is likely you ran into a a buggy version. In these cases, don't feel bad about passing `lax=True`.

There are some legacy attributes and methods that you should no longer use: `access_url`, `service`, `search()`; all these only do something sensible when there is only one capability on a `RegistryResource`. This is not unlikely if you did constrain the `servicetype` in your call to `search`. But in general it is a much better idea to search for data and decide on access modes later in data discovery. Most resources today come with multiple capabilities, and it is good if you can choose the most appropriate for your task at hand.

Interactive Use of the PyVO Registry API

Finally: A jupyter notebook!

Ψ data-discovery-demo.ipynb



Exercise 12

Can you figure out the default output limit (i.e., in effect an implied TOP) for the TAP service at http://dc.g-vo.org/tap? How far can you raise it?

Can you write a program that figures it out for all TAP services out there that talk about tgas?

Exercise 13

Which IAU constellation is the least massive exoplanet in the exoplanet merged catalogue in? Try solving this using pyVO's registry API; hint: to figure out constellations, having the constellations as ADQL polygons is really handy.

Resolving Ivoids

IVOA identifiers are the primary keys in the VO Registry.

When keeping notes like "which service did I use", the ivoid (rather than a DOI) still is the better choice in the VO for the simple reason that all VO resources have an ivoid, but many have no DOI.

To resolve an ivoid:

```
svc = pyvo.registry.search(ivoid='ivo://org.gavo.dc/tap')[0]
```

You can then go on as we did above with `access_modes`, `get_service`, etc.

Write Your Own Constraint

registry.search uses constraint classes to build queries.

You can extend the set of constraint classes yourself by inheriting from `registry.SubqueriedConstraint`.

Say you want to use the experimental UAT extension to RegTAP, i.e., `rr.uat_concept`:

```
class UATConcept(pyvo.registry.SubqueriedConstraint):
    _keyword = "uat"
    _subquery_table = "rr.subject_uat"

    def __init__(self, uat_id):
        self._condition = "uat_concept={uat_id}"
        self._fillers = {"uat_id": uat_id}
```

Ψ new-constraint.py

What is going on here?

- We define a class inheriting from the base class `SubqueriedConstraint`. This is defined in pyvo/registry/rtcons.py; but the code there is rather dense, so it is probably best to look at other classes that are SubqueriedConstraint-s further down the source to get a feeling for how this is supposed to work.
- The first thing we need to define is what table we want to match in; this ends up in the `_subquery_table` class variable. Here, we are using an extension on the default RegTAP server, a table containing UAT keywords for all the services. This is more useful than what is in the standard `rr.res_subject` table, as there, you have all kinds of words and keyword schemes and all that – but the UAT table is non-standard, which may be the reason why you need to write your own constraint.
- In the constructor, we fill instance attribute `_condition`, which needs to contain ADQL suitable for WHERE. However, this is just a template with fields (here: the stuff with curly braces) to be replaced when the machinery bakes the actual query.
- For each template field, we have to give a key-value pair in the `_fillers` dictionary. Here, there is just `uat_id`. The reason this is done behind the scenes is that we want to make SQL string literals from python strings, and the logic to do that should not be repeated in each constraint class but in only one central place.
- The rest of the query generation is done by pyvo.registry. In reality, this is often a bit more complex, for instance, because you may want to have multiple terms combined with OR; when you pass multiple constraints, they are combined with AND. See, for instance, the UCD constraint for how you would go about this.
- The `_keyword` class variable gives the name of the keyword argument equivalent to passing in a `UATConcept` constraint.

Exercise 14

(You will need to have looked at the vocabularies sidetrack for this)

Take new-constraint.py and add support for query expansion: add a keyword argument `expand`. If that is true, include the narrower concepts of what was passed in, too.

Hint: You can leave (something like) this to the server with a UDF, or you can do the query expansion locally; the first way is simpler, the second perhaps more instructive.

5 Datalink

Datalink: Getting Related Artefacts

Datalink is a standard for "linking" files to datasets. Think calibration data, previews, extracted objects, alternative formats, etc.

https://dc.g-vo.org/static/datalinks.shtml is a showcase of various applications of datalink. You can retrieve the links in a web browser and ought to get a reasonable UI if you have enabled javascript.

This is really machine-readable data; load any of these links into TOPCAT to inspect it as a VOTable:

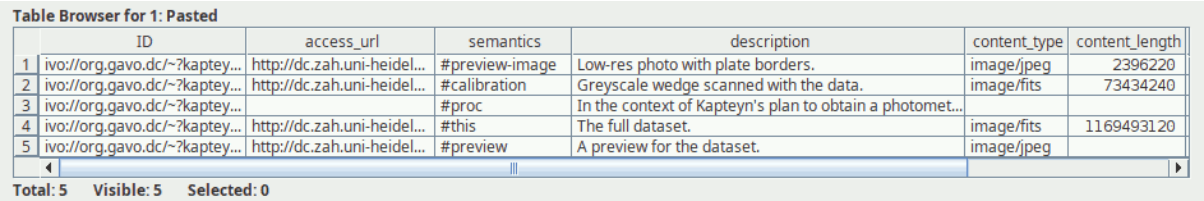


Table Browser for 1: Pasted

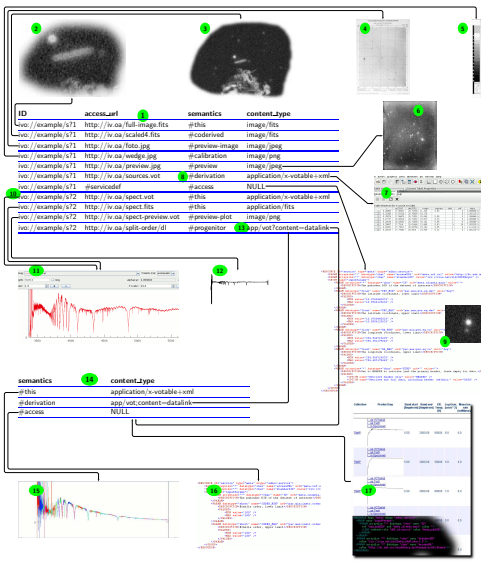
	ID	access_url	semantics	description	content_type	content_length
1	ivo://org.gavo.dc/~?kaptey...	http://dc.zah.uni-heidel...	#preview-image	Low-res photo with plate borders.	image/jpeg	2396220
2	ivo://org.gavo.dc/~?kaptey...	http://dc.zah.uni-heidel...	#calibration	Greyscale wedge scanned with the data.	image/fits	73434240
3	ivo://org.gavo.dc/~?kaptey...		#proc	In the context of Kapteyn's plan to obtain a photomet...		
4	ivo://org.gavo.dc/~?kaptey...	http://dc.zah.uni-heidel...	#this	The full dataset.	image/fits	1169493120
5	ivo://org.gavo.dc/~?kaptey...	http://dc.zah.uni-heidel...	#preview	A preview for the dataset.	image/jpeg	

Total: 5 Visible: 5 Selected: 0

The power of datalink comes from the fixed structure of these rows, which allows machines to do sensible things with them. The rows (normally) consist of

- a (theoretically globally unique) ID of the dataset the link is for
- a URL for the data linked `access_url`
- a human-readable `description`,
- `semantics`, that is, a machine-readable identification of what this link is. This comes from a controlled vocabulary, http://www.ivoa.net/rdf/datalink/core. This allows clients to sensibly group and/or select these links
- a type and length of the content that lets client figure out what to do with the file: `content_type`, `content_length`
- and a few more technical fields.

Datalink in a Cartoon



This is a cartoon of a datalink response for two different datasets, a scanned plate and a spectrum; that datalink allows you have links for multiple datasets in one document is something I would consider a misfeature more than anything else, but for this example it was convenient. If you zoom in sufficiently far, you can spot the following:

The first seven rows in correspond to a scanned plate. There is a placeholder for the **original dataset** with semantics `#this`, i.e., the "main" dataset. A **rebinned version** (the figure shows a larger area) is declared as `#coderived` from the main dataset. The semantics here could be a bit more precise to indicate this link is just the resampled `#this`. If there were a clear idea what a machine would do differently if it knew that, one can define a refined term using IVOA processes (look for "IVOA VEP" if interested).

The original plate was part of an early survey which has been published in book form. A JPEG **photo of the book page** corresponding to the plate is declared as `#preview-image` in row three.

Datalink is ideal for declaring files from a dataset's provenance chain. In row four, we include a **PNG grey wedge** from the scan with `#calibration` semantics.

In the other direction, you can also declare derived data products, such as the sources.vot in row five, supposed to be a **table of extracted sources** from the image; the corresponding semantics is `#derivation`, and again there may be cases when some more refined term for extracted sources would be beneficial and should be defined.

Row six has a **thumbnail** of the image, declared as a `#preview`.

The next row defines a **cutout service**. Datalink allows a straightforward declaration of the parameters for server-side data manipulation services within the VOTables that return datalink metadata. If you decipher the XML, you will see that this is sufficient not only to operate the service but also produce attractive UIs by declaring units, UCDs, and ranges of the pertinent parameters.

The remaining three rows correspond to a spectrum (a single datalink document can contain links for more than one dataset, but in practice that is rare).

The semantics `#this` in row eight should already be familiar; it corresponds to a **spectrum** here.

The **preview** in spectrum case is a plot, which is reflected in the different semantics. A client consulting the datalink vocabulary will figure out that `#preview-plot` actually is-a `#preview`.

The last datalink shows **recursive datalink**: its file has the media type

application/x-votable+xml;content=datalink

that designates datalink documents (and can be used in protocols like ObsTAP, too). In this case, the datalink is for a `#progenitor` in the provenance chain, which here is a file with unmerged Echelle orders.

Datalink in PyVO

In pyVO, datalink is (primarily) exposed in search results.

On datalink-enabled services, you can iterate over `iter_datalinks()`, which iterates over `DatalinkResults` instances.

On these, you can pull links using `bysemantics`:

```
svc = pyvo.ssa.SSAService("http://dc.g-vo.org/feros/q/ssa/ssap.xml?")
matches = svc.search(SkyCoord.from_name("EI Eri"), 0.001)

for links in matches.iter_datalinks():
  for link in links.bysemantics("#preview"):
    print(link["access_url"])
```

Or just iterate over `links` to see all links available.

Yes, this is a bit deeply nested in the way of iteration, but that is the price of flexible protocols. The links come as dictionary-like objects with keys matching the column labels from the datalink specification. The labels are those written in typewriter in the enumeration of the datalink fields above.

> **Exercise 15**
>
> Write a function `get_available_semantics(dl) -> set` returning a set of the semantics available for a given datalink.
>
> Try your program on the SSA example from the lecture.

Use Case: Overview With Previews

Let's say you want to spot bad or weird spectra without actually retrieving or plotting the spectra themselves.

Just download the previews and merge them into one image:

```
svc = pyvo.ssa.SSAService("http://dc.g-vo.org/feros/q/ssa/ssap.xml?")
matches = svc.search(SkyCoord.from_name("EI Eri"), 0.001)
previews = []
for dl in matches.iter_datalinks():
    prev_url = next(dl.bysemantics("#preview"))["access_url"]
    im = Image.open(io.BytesIO(requests.get(prev_url).content))
    previews.append(im)
```

Ψ datalink-previews.py

The perhaps slightly alarming `next(...)` construct is just "pick off the first item from an iterator"; we can do that here because we only want one preview per dataset (and actually, there is only one). This is a convenient construct when dealing with the nested iteration in datalink in many cases when you (think you) know there is only one link with a certain semantics.

The full source has some code merging all the previews into one raster image using the excellent python imaging library PIL.

Datalink: Remote Processing on Datalink Documents

Datalink also lets you declare processing services. The SODA standard defines a special set of parameters applicable to astronomical images (CIRCLE, POLYGON, TIME, BAND,...).

Save *a lot* of time by only downloading cutouts of the object you are interested in:

```
roi = SkyCoord.from_name('Mira')
for rec in svc.run_sync(
      "SELECT access_url, access_format FROM ivoa.obscore"
      " WHERE obs_collection='HDAP'"
      "AND 1=CONTAINS(CIRCLE('ICRS', {}, {}, 0.05),"
      "s_region)".format(roi.ra.deg, roi.dec.deg)):
    processed = rec.processed(
        circle=(roi.ra.deg, roi.dec.deg, 0.05))
```

Ψ datalink-soda.py

This example retrieved datasets that come as datalinks directly. This is why we included `access_format` in the obscore query: This way, pyVO knows when it is dealing with a datalink document, and it will add the `iter_datalinks` and `processed` methods when the service offers the necessary facilities.

It is more common to deliver "normal" files and offer datalink on the side. In this case, things get somewhat more complicated at the moment because with the current API you can *either* see the actual records or the datalinks.

Datalink: Remote Processing on Non-Datalink Documents

Use case: Hα maps of Sd galaxies from CALIFA.

CALIFA is a collection of spectral cubes (i.e., an array of small-band images) of galaxies; there is a datalink-enabled TAP table (califadr3.cubes) listing the cubes on the TAP service http://dc.g-vo.org/tap. We can extract Hα maps by doing spectral cutouts, supported via SODA's BAND parameter (which takes vacuum wavelengths in meters).

Use TOPCAT to inspect the tables belonging to califadr3; in particular note the objects table that you can join with cubes via the califaid column. The cubes come in three different setups. To avoid duplicate data, we will only look at COMB data.

Hα is at 656.25 nm (vaccuum) in the lab. For the low redshifts we are talking about here, $\lambda_{\mathrm{lab}} = (1+z)\lambda_0$ is just fine to compute where the galaxy's Hα is at the spectrograph.

Doing the cutouts by calling `processed` on the link for the data itself (`#this`):

```
matches = svc.run_sync(
    "SELECT califaid, obs_publisher_did, mime, em_min, em_max, redshift"
    " FROM califadr3.cubes"
    " JOIN califadr3.objects USING (califaid)"
    " WHERE setup='COMB' AND hubtyp='S d'")

for dl in matches.iter_datalinks():
    lobs = ???
    map = next(dl.bysemantics("#this")).processed(band=(lobs, lobs))
```

Trouble: How do I find the redshift (i.e., `lobs`) for my `dl`?

The (current) answer is: Use `ID` in the dl rows to match against `obs_publisher_did` in `matches`. How do you know it's that column? Well, for obscore and obscore-like tables, it will almost always be that.

If you have to dig yourself, things get messy because pyVO does not expose that information properly yet. Meanwhile, you can trudge on by inspecting the VOTable. You first get the service definition for the cutout service, most of the time the first service there is (in VOTable, that corresponds to a RESOURCE). In there, look at the PARAMs of the GROUP in there, and you will find a PARAM named `ID`. Whatever is in its `ref` attribute is what you are looking for:

```
>>> svc = next(matches.iter_adhocservices())
>>> print(list(svc.groups[0].iter_fields_and_params()))
[<PARAM ID="ID" arraysize="*" datatype="char" name="ID"
  ref="obs_publisher_did" ucd="meta.id;meta.main" value=""/>]
```

Yes. There should be a better and more robust API for this; in pyVO 1.6, there you will probably have an `original_row` attribute on what you get back from `iter_datalinks`.

Datalink: Simultaneous Links and Metadata

```
matches = svc.run_sync(
    "SELECT califaid, obs_publisher_did, mime, em_min, em_max, redshift"
    " FROM califadr3.cubes"
    " JOIN califadr3.objects USING (califaid)"
    " WHERE setup='COMB' AND hubtyp='S d'")
result_rows = matches.to_table()
result_rows.add_index("obs_publisher_did")

for dl in matches.iter_datalinks():
    rec = result_rows.loc["obs_publisher_did", dl["ID"][0]]
    califaid = rec["califaid"]
    lobs = l0*(1+rec["redshift"])
    processed = next(dl.bysemantics("#this")
      ).processed(band=(lobs, lobs))
```

Ψ soda-with-rows.py

The novelty here is that we are making a proper astropy table of the results now in order to be able to create an *index* on it. That's a technial term for "make it so we can fetch rows quickly by using values from this column". With the `add_index` call, we can use `.loc` attribute on the table to quickly pick out rows by `obs_publisher_did`. This is how we can find the table row for a datalink.

Exercise 16

Get the soda-with-rows.py script for doing cutouts on CALIFA DR3 and make a false colour image for IC 1151 by taking the slices from the COMB cube (see the setup column) at 400 nm as blue, at 550 nm as green, and at 700 nm as red. Do not download the whole cube, use SODA to just retrieve exactly what you need.

Hint: If you have no better way to combine single-channel pixels to an RGB image in Python, use the excellent Python Image Library PIL (in its modern incarnation of pillow). This is still not entirely trivial, so here is how to get three arrays red, green, and blue, made up of three frames into a colour jpeg using plain PIL and numpy:

```
def _normalize_for_image(pixels):
  pixels = numpy.flipud(pixels)
  pixMax, pixMin = numpy.max(pixels), numpy.min(pixels)
  pixels = (pixels-pixMin)/(pixMax-pixMin)*255
  return numpy.asarray(pixels, numpy.uint8)

pixels = numpy.array([
    normalize_for_image(red),
    normalize_for_image(green),
    normalize_for_image(blue)]).transpose(1,2,0)
Image.fromarray(pixels, mode="RGB"
  ).save("IC1151.jpeg", format="jpeg")
```

I have not tried looking for a less pedestrian way to do this; if you have one, please write in.

6 Higher SAMP Magic

Use Case: An Object Investigator

Let's say you are debugging your pipeline and want to manually inspect "weird" objects by querying a set of other catalogues have on them.

Plan: Write a program that other clients

- can send tables to and then
- when a table row is selected, computes a new table with data from other services
- that is then sent to Aladin for inspection.

SAMP: Listening to Messages

SAMP is based on messages; there are several message types (*MType*-s), which are documented on the IVOA wiki[14].

The SAMP client objects's `bind_receive_message` method arranges for the hub to call a function when a message of a certain MType comes in. The calling pattern is a bit complicated, but what really counts is a dictionary of the parameters passed to the call on the sender side (`params`).

SAMP has two types of messages: Notifications, which do not expect a response, and calls, which do. If you use `bind_receive_message`, you will cover both cases, which is generally a good idea, because all kinds of messages can come as either.

[14] http://wiki.ivoa.net/twiki/bin/view/IVOA/SampMTypes

If a call (as opposed to a notification) comes in, it is associated with a message id, and the sending client will expect a response. If you do not give one, you will have ugly "pending" SAMP messages. Notifications have no message id, and they require not responses.

Here is a program that prints sky coordinates of "things" the user pointed to:

```
import pyvo
import vohelper

@vohelper.show_exception
def print_coord(privkey, sender_id, msg_id, mtype, params, extra):
  print("{} {}".format(params["ra"], params["dec"]))
  if msg_id is not None:
    conn.reply(msg_id, {"samp.status": "samp.ok", "samp.result": {}})

with pyvo.samp.connection(addr="localhost") as conn:
  conn.bind_receive_message("coord.pointAt.sky", print_coord)
  input()
```

The handler function has a rather complex *signature* (i.e., what parameters it takes and what it returns). Don't sweat it too much. In particular, do not be alarmed when you ignore `private_key`; for all I know no client at this point does any kind of cryptographic validation. There *are* security implications from SAMP, but very frankly: if you regularly have your browser execute Javascript from random web pages, you are in worse trouble.

The important part is `params`; this is where the parameters given on the SAMPMtypes page are in; in the case of the *coord.pointAt.sky* message we receive here, these are in the keys *ra* and *dec*. To try this, start Aladin and then the sample program. When you click on the sky, you will see the target coordinates in your terminal.

Versus the basic "Add SAMP Magic" method of getting a SAMP connection, we have now added an `addr="localhost"`. This is a workaround to make listening to messages a bit more robust on machines that have both IPv4 and IPv6 enabled (most have in 2024). If you get "connection refused" messages or the like when trying to send a message, try removing the argument.

As said above, when `msg_id` is not None (i.e., we got a call, not a notification), we have to send a reply. The sample code essentially says: "I have no results, and that is fine for this MType".

MTypes for the Vicinity Searcher

To make our program ready to receive tables via SAMP, we have to listen to *table.load.votable*. Params for that as per the MTypes wiki page:

url URL of the VOTable document to load

table-id local identifier for referencing

name human-readable name

To monitor whether a row in a table you received is selected, listen to *table.highlight.row*. Params:

table-id the local identifier

row the row index

Python Classes: Why?

We have to keep quite a bit of state in our program, at least:

- the SAMP connection
- the table sent to us.

There is also quite a bit of behaviour:

- receive and store the remote table
- see when rows are selected
- do searches when that happens.

When you have state and behaviour linked together, in Python think: "class".

Python Classes: How?

```
class VicinitySearcher:
  vicinity_size = 30
  client_name = "Aladin"

  def __init__(self, conn):
    self.conn = conn
    self.cur_table = self.cur_id = None

  def load_VOTable(self,
    private_key, sender_id, msg_id, mtype, params, extra):
    ...

  def handle_selection(self,
    private_key, sender_id, msg_id, mtype, params, extra):
    ...
```

Ψ vicinitysearcher.py

The trivial version of object lore in python is: All functions belonging to an object (*method*-s) have a first argument conventionally called `self` (the *instance*), and whenever you put an attribute on `self`, you can find it again in other methods' `self`, provided these other methods are called on the same *instance* (i.e., object).

You can also have attributes in the class itself; consider these constants, as assigning to these may not always do what you expect.

To call other methods of the same object, use `self.methodname()`.

Create an instance by calling the class (here: `VicinitySearcher(conn)`). Whatever you pass into that call will be passed to the `__init__` method (the *constructor*).

Handling table.load.votable

```
class VicinitySearcher:
  def __init__(self, conn):
    [...]
    self.conn.bind_receive_call(
      "table.load.votable", self.load_VOTable)

  def load_VOTable(self,
    private_key, sender_id, msg_id, mtype, params, extra):
    self.cur_table = Table.read(params['url'])
    self.cur_id = params["table-id"]
    self.conn.reply(msg_id,
      {"samp.status": "samp.ok", "samp.result": {}})
```

Since we bind the SAMP *table.load.votable* MType to **self**.load_VOTable (a *bound method*, which **VicinitySearcher**.load_VOTable would not be), we get our instance of VicinitySearcher (`self`) passed into our method for free.

When we then get notified of a table load, we set some instance variables that let us work with the table later.

To make this robust, we should catch exceptions and send replies with a status of `samp.error` in case of trouble; as said above, clients really want *some* reply when they send messages directly to clients and complain about pending SAMP calls when they receive none.

Handling table.highlight.row

```
  @vohelper.show_exception
  def handle_selection(self,
    private_key, sender_id, msg_id, mtype, params, extra):
    if params["table-id"]!=self.cur_id:
      return
    table_index = int(params["row"])
    print("Row selected:", table_index)
    response = self.make_response_table(table_index)

    if response is not None:
      vohelper.send_table_to(self.conn, self.dest_client, response)
```

The `@vohelper.show_exception` thing before the method definition is called a *decorator*. These are things (actually: functions) that operate on methods. In this particular case, all it does is make sure any exceptions raised within the SAMP handler are properly displayed. Since the SAMP handlers do not run in the main thread (and thus exceptions do not terminate the program), without this you will miss errors in the handlers.

The actual functionality (in this case, searching for matching data in a few catalogues and broadcasting any matches found) I have delegated to another method, `make_response_table`. This is an example for using Simple Cone Search; have a look at it!

> **Exercise 17**
>
> The action of the SAMP handler is in the `make_response_table` method; have a brief look at it to appreciate what is going on. Then, replace what is there with something that does a SIAP search on the service at http://dc.g-vo.org/lswscans/res/positions/siap/siap.xml and returns the corresponding table for sending to Aladin (hint: remember the `to_table` method of DAL results).

Exercise 18

Listening to the SAMP message *coord.pointAt.sky*, implement an "odometer" computing and printing after each step the distance travelled by the pointer.

To do this, you will need to keep the SAMP connection, the last position and the distance travelled so far as state; take the vicinitysearcher, remove the code keeping the state and behaviour used for its function, and insert our new logic.

Hints: Look at SkyCoord in Astropy and the mtypes page; when re-using SAMP bindings, make sure you handle messages, not calls.

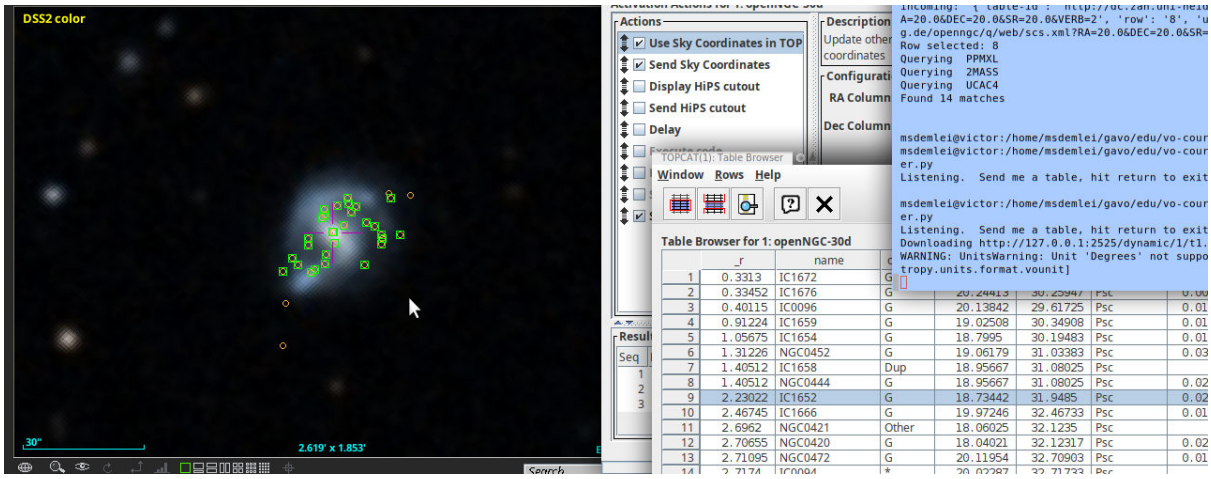
Try It Out

Start TOPCAT, Aladin, and the vicinity searcher.

Look for openngc SCS and pull some 40 degree cone.

Send the resulting table to the vicinity searcher, have *Send row index* as an activation action.

Click on table rows or plot points.



	_r	name					
1	0.3313	IC1672	G	20.24413	30.25947	Psc	0.00
2	0.33452	IC1676	G	20.13842	29.61725	Psc	0.01
3	0.40115	IC0096	G	19.02508	30.34908	Psc	0.01
4	0.91224	IC1659	G	18.7995	30.19483	Psc	0.01
5	1.05675	IC1654	G	19.06179	31.03383	Psc	0.03
6	1.31226	NGC0452	G	18.95667	31.08025	Psc	
7	1.40512	IC1658	Dup	18.95667	31.08025	Psc	0.02
8	1.40512	NGC0444	G	18.73442	31.9485	Psc	0.02
9	2.23022	IC1652	G	19.97246	32.46733	Psc	0.01
10	2.46745	IC1666	G	18.06025	32.1235	Psc	
11	2.6962	NGC0421	Other	18.04021	32.12317	Psc	0.02
12	2.70655	NGC0420	G	20.11954	32.70903	Psc	0.01
13	2.71095	NGC0472	G				

7 At the Limit: VO-Wide TAP Queries

VO-Wide TAP Queries

People often say: "I want everything in the VO on object X".

This is far too hard. There are many reasons why this is hard, beginning with what "everything" is – for instance, you would not normally want every frame containing the object ever taken.

What *is* marginally possible: "Give me all measurements of a certain sort of UCD in a certain vicinity." Actually, the constraints can be a lot more general than just a cone search, as long as you can formulate it with UCDs.

However, this is surprisingly involved, mostly for stupid reasons. Follow me along for proper motions (`pos.pm`).

Note: This is probably not something realistic for research within the next few years. But it is a nice exercise in how far you can take pyVO and TAP.

A RegTAP Query for Tables and TAP Services

For "where can I find data with UCD X?", there is `pyvo.registry.UCD`.

But we need to know *which table* has a column with our UCD.

PyVO can't do that yet; hence, use a direct RegTAP query:

```
SELECT DISTINCT access_url, table_name
FROM rr.interface
NATURAL JOIN rr.capability
NATURAL JOIN rr.res_table
NATURAL JOIN rr.table_column
NATURAL JOIN rr.stc_spatial
WHERE
    standard_id LIKE 'ivo://ivoa.net/std/tap%'
    AND ucd LIKE 'pos.pm%'
    AND 1=INTERSECTS(POINT({RA}, {DEC}, {SR}), coverage)
    AND (table_type!='output' OR table_type IS NULL)
```

How do you come up with a query like this? Well: you *can* start from what pyVO does; pyvo.registry has the `get_RegTAP_query` function that will return what pyVO would generate for a given set of constraints. For instance:

```
import pyvo

print(
  pyvo.registry.get_RegTAP_query(
    pyvo.registry.UCD('pos.pm%')))
```

outputs this horror:

```
SELECT
ivoid, res_type, short_name, res_title, content_level, res_description,
reference_url, creator_seq, created, updated, rights, content_type,
source_format, source_value, region_of_regard, waveband,
  ivo_string_agg(COALESCE(access_url, ''), ':::py VO sep:::') AS access_urls,
  ivo_string_agg(COALESCE(standard_id, ''), ':::py VO sep:::') AS standard_ids,
  ivo_string_agg(COALESCE(intf_type, ''), ':::py VO sep:::') AS intf_types,
  ivo_string_agg(COALESCE(intf_role, ''), ':::py VO sep:::') AS intf_roles,
  ivo_string_agg(COALESCE(cap_description, ''), ':::py VO sep:::') AS cap_descriptions
FROM
rr.resource
NATURAL LEFT OUTER JOIN rr.capability
NATURAL LEFT OUTER JOIN rr.interface
NATURAL LEFT OUTER JOIN rr.alt_identifier
NATURAL LEFT OUTER JOIN rr.table_column
WHERE
(ucd LIKE 'pos.pm%')
GROUP BY
ivoid, res_type, short_name, res_title, content_level, res_description,
reference_url, creator_seq, created, updated, rights, content_type,
source_format, source_value, region_of_regard, waveband
```

This is massively uglified by pyVO's need to be generic and to, in a single query, pull all kinds of information on the services available. In tailored RegTAP queries you rarely need that kind of thing. Still, you could take this query and strip it down until it does what you want, in particular as regards what tables to hit in the first place.

Alternatively, RegTAP is written such that to build a query, you only have to look for what table a piece of data you want to retrieve or constrain is in and then NATURAL JOIN with the table.

The canonical source to find this kind of information is the RegTAP standard, Demleitner and Harrison et al. (2019), in particular its Figure 2; also skim over the example queries in section 10 if you need to hand-write RegTAP queries.

In this case, to be able to query TAP services, we need the access url (in rr.interface) of the service and the table name (in rr.res_table) for a table containing a column with a UCD (in rr.table_column). To be able to say "I want a TAP service", we need to constrain the standard identifier (in rr.capabilty). Finally, we want to throw out tables that do not have data for our region of interest, and hence we also need to constrain the spatial coverage (in rr.stc_spatial). That consideration almost results in the hand-tailored query shown above already.

Two details are in there on top: the DISTINCT after the SELECT is so we do not get one pair of access url and table name for every *column* in the tables that have matching UCDs; in general, there will be more than one of them, and we still only want to query the table once.

And then there is the odd

```
AND (table_type!='output' OR table_type IS NULL)
```

This is another instance of where something seemed like a good idea to the standards designers – in this case: Use the same elements to declare output tables and queriable tables – makes for something that is hard to understand in later use. What this means is: Ignore tables that are declared in the registry but that one probably cannot query.

Running the RegTAP Query

Running RegTAP queries just means picking a suitable TAP service and calling `run_sync`:

```
reg_svc = pyvo.registry.regtap.get_RegTAP_service()
result = reg_svc.run_sync(regtap_query)

svcs = {}
for row in result.to_table():
  svcs.setdefault(row["access_url"], []).append(row["table_name"])
return svcs.items()
```

There is no magic behind `get_RegTAP_service` – it is constructing a normal `TAPService`, just configured with an access URL known to lead to a RegTAP service. By the way, you can change that URL if you want to use a different registry service; use `choose_RegTAP_service` from within pyVO, or set the `IVOA_REGISTRY` environment variable to your preferred RegTAP service's access URL.

Note how we are grouping the tables belonging to a service in this code. This is exactly a GROUP BY operation in the database sense. So:

> **Exercise 19**
>
> In multitap.py, have a look at `get_services_and_tables`; in there, we are doing a grouping operation on the client (i.e., our) side. Can you move to to the server side using GROUP BY and the `ivo_string_agg` UDF?

Query Generation I: Defining the Schema

We want to build queries that let us fill a table defined like this:

```
#    col-name,  UCD,                 Unit,    type-to-cast-to
RESULT_SCHEMA = [
    ('cat_id', "meta.id;meta.main",   None,     "CHAR(*)"),
    ('ra',     "pos.eq.ra;meta.main", "deg",    None),
    ('dec',    "pos.eq.dec;meta.main","deg",    None),
    ('pmra',   "pos.pm;pos.eq.ra",    "mas/yr", None),
    ('pmde',   "pos.pm;pos.eq.dec",   "mas/yr", None),]
```

You may recognise our technique of writing "inhomogeneous" records in tuples from our fetch3 example. In this case, we give names, the UCDs from which to fill the columns, the target unit, and a type the column should have; this is important in the case of `cat_id`, the object identifier within the catalogue, which sometimes is an integer and sometimes is a string. We have to unify this if rows from different tables are supposed to end up in one result table. All other columns will be real-valued if they are somehow sane, and hence we do not need to cast.

We now need to write code that can create database queries from these specifications and table metadata.

Query Generation II: From Clause And a Template

Given a TAP service `svc`, a `table_name`, our result schema, and the region of interest in RA, DEC, and SR, make a query to produce rows for our result schema:

```
db_table, select_clause = svc.tables[table_name], []
for dest_name, ucd, unit, type in RESULT_SCHEMA:
    select_clause.append("{} AS {}".format(
        fieldname_with_ucd(ucd, db_table),
        dest_name))
select_clause.append(f"'{table}' AS table_name")
select_clause.append(f"'{svc.baseurl}' AS svc_url")

return ("SELECT {select_serialised} FROM {srctable}"
    " WHERE 1=CONTAINS(POINT('ICRS', {racol}, {deccol}),"
    "   CIRCLE('ICRS', {ra}, {dec}, {sr}))").format(
        select_serialiased=", ".join(select_clause),
        srctable=table_name,...)
```

In this snippet, we first incrementally build a select clause by looking for the UCDs we are interested in in the remote table definition (that we retrieve using `svc.tables[table_name]`) and make "their-name AS our-name" particles. We add two constant fields for the table name and the service access URL; this is so we can later still see where everything came from.

We have to define the function `fieldname_with_ucd` ourselves, because astropy tables (which is what is in `svc.tables`) do not have the convenient `fieldname_with_ucd` method that pyVO `DALResults` have. Perhaps this should change? Anyway: the implementation is trivial, except that we lowercase both the incoming UCD and the UCDs we get from the service. Curse case-insenstitive items.

These particles are then joined into the selclause in the ADQL template.

Query Generation III: Delimited Identifier Workaround

Regrettably, the code immediately fails.

```
$ python3 multitap-broken1.py
[...]
pyvo.dal.exceptions.DALQueryError:
    Incorrect ADQL query:
    Encountered "/". Was expecting one of: <EOF> "." "," ";" "AS"
        "WHERE" "GROUP" "HAVING" "ORDER" "\""
        <REGULAR_IDENTIFIER_CANDIDATE> "NATURAL" "INNER" "LEFT"
        "RIGHT" "FULL" "JOIN"
```

Ψ multitap-broken1.py

The problem: Vizier uses delimited identifiers but has them unquoted in the registry. Workaround:

```
def perhaps_quote(table_name):
    parts = table_name.split(".")
    for index, part in enumerate(parts):
        if not re.match("[A-Za-z0-9][A-Za-z0-9_]*$", part):
            parts[index] = '"{}"'.format(part.replace('"', '""'))
    return ".".join(parts)
```

This is a long-standing problem on VizieR's side; the standard has been clear on this for a long time ("when delimited identifiers are used as table names on the relational side, the quotes must be part of name's value, and the capitalisation used in the DDL must be preserved"), and actually, a function like `perhaps_quote` cannot even really work (e.g, in "USNO-B-1.0", is the dot part of a name or a schema separator?). So – this is another illustration of where sometimes one has to live with imperfections and just cope as well as possible.

Running Queries I: Feature Detection

On a service like VizieR with our *pos.pm* criterion, we will have to query a lot of tables and stack the results on the client side.

Don't take my word for "a lot of tables"; on VizieR, at the time of writing, the ADQL query

```
SELECT COUNT(*)
FROM (
    SELECT DISTINCT table_name FROM tap_schema.columns
    WHERE ucd LIKE 'pos.pm;%') AS q
```

returns a whopping 2003; in reality, due to our positional constraint, we would be firing off a lot fewer queries, but it would still be nice if we only had to run one.

Can we take a union of the results on the server side?

Perhaps. We need the ADQL UNION operator for that. Regrettably, it is optional.

Interactively, you will find information on supported features in the *ADQL* tab of modern TOPCATs. From within pyVO, there is a complex hierarchy of objects below a TAPService instance. Unless you really want to read Demleitner and Dowler et al. (2012), take the following blindly as a recipe.

Does a service support union?

```
knows_union = svc.get_tap_capability().get_adql().get_feature(
   "ivo://ivoa.net/std/TAPRegExt#features-adql-sets", "UNION")
```

The `get_feature` method takes two arguments: An ivoid identifying the sort of feature you are querying, and a key into the feature listing. To figure out these two strings, I am afraid you will need to consult the ADQL standard (Mantelet and Morris et al., 2023).

Incidentally, as of mid-2024, VizieR's ADQL engine does not yet support UNION, which is the main reason we have put it a sanity break in multitap.py

```
    if len(tables)>30:
        sys.stderr.write("  (cropping to 30 tables for handleability)\n")
        tables = tables[:30]
```

(but then we probably would have anyway, because even a union over 300 tables is a bit too much for an educational example).

Running Queries II: Adapting to Server Capabilities

Since UNION is optional, we have to have two code paths now, one for services with UNION, one for ones without. It will not get much simpler than that:

```
svc = pyvo.dal.TAPService(access_url)
knows_union = svc.get_tap_capability().get_adql().get_feature(
    "ivo://ivoa.net/std/TAPRegExt#features-adql-sets", "UNION")

queries = [get_query(svc, table_name)) for table_name in tables]

result_rows = []
def feed_rows(astropy_table):
    for row in astropy_table:
        result_rows.append(dict(zip(row.colnames, row.as_void())))

if knows_union:
    feed_rows(svc.run_sync(
        " UNION ".join(queries)).to_table())
else:
    for query in queries:
        feed_rows(svc.run_sync(query).to_table())
```

This is seriously ugly code; to smuggle shared code into the two legs of the `knows_union` selection, we first take out the generation of the queries from where they run, and then create a local function encapsulating the logic of processing result rows (this is called a *closure* in this case, because the function encloses the `result_rows` list from the parent block).

And we have two rather different pieces of code on the two sides of the selection. They will age and break differently, and all this is painful.

Take it from me: Optional features suck. In almost everything. If you ever write software or a standard, try to avoid them as much as you can.

> **Exercise 20**
>
> Can you find out the strings you need to pass to `get_feature` find find out whether a service supports the nifty `IN_UNIT` function?

Query Generation IV: Casting

Even this ends with an obscure error. Try multitap-broken2.py

Ψ multitap-broken2.py

```
pyvo.dal.exceptions.DALQueryError: Field query: UNION types integer
and text cannot be matched LINE 1: ...S(12), RADIANS(13)), RADIANS(0.1))))
UNION SELECT localid AS...
```

The reason? Idenifier columns are sometimes integers and sometimes texts.

The solution? Cast them all to string.

But: CAST is optional. Oh no!

We could probably get away with just blindly casting, because as long as a service does not support UNION, we can do the casting locally in Python, while services with UNION will probably support CAST, too. But that's just guessing, and this is more about education rather than economy of work.

Query Generation V: Still Casting

```
knows_cast = svc.get_tap_capability().get_adql().get_feature(
            "ivo://ivoa.net/std/TAPRegExt#features-adql-type", "CAST")

for dest_name, ucd, unit, type in RESULT_SCHEMA:
    if type and knows_cast:
        select_clause.append("CAST({} AS {}) AS {}".format(
            perhaps_quote(fieldname_with_ucd(ucd, db_table)),
            type,
            dest_name))

    else:
        # Don't cast and hope for the best
        select_clause.append("{} AS {}".format(
            perhaps_quote(fieldname_with_ucd(ucd, db_table)),
            dest_name))
```

The fallback is of course error-prone: If a table schema would need a CAST but the service cannot do it, we may fail that service. Sometimes best effort is all one can do.

Bringing it all together

After all this preparation, the actual program is trivial except for our usual error handling:

Ψ multitap.py

```
recs = []
svcs_and_tables = get_services_and_tables()
for svc_url, tables in svcs_and_tables:
    try:
        recs.extend(get_rows_for_svc(svc_url, tables))
    except Exception as msg:
        import traceback; traceback.print_exc()
        sys.stderr.write(f"{svc_url} broken (skipped): {msg}\n")

res_table = make_result_table(recs)
res_table.write("all-pms.vot", format="votable", overwrite=True)
with pyvo.samp.connection() as conn:
    pyvo.samp.send_table_to(conn, res_table,
        name="all-pms", client_name="topcat")
```

Exercise 21

There is one glaring hole in our multitap script: Units. Try to improve on this: If the service supports `IN_UNIT`, use it in about the way we have been using CAST.

If you actually need something like this, you can of course also compute the conversion factors locally (using astropy.units) and bake them into the queries. Feel free to try that, too.

8 Odds and Ends

8.1 EPN-TAP

EPN-TAP 1

EPN-TAP is like obscore, just for solar system data. That is: there is a pre-defined schema that you can query on many services in a uniform way. normal VO TAP plus a pre-defined table structure; the tables are always called `epn_core`. Columns of note include:

- `granule_uid` – an identifier for the dataset ("granule" is a word for something like a dataset in solar system sciences).
- `target_name` – what was observed? Regrettably, there are no strict rules for what is called what, so it requires a certain amount of domain feeling to guess how to constrain this.
- `time_min`, `time_max` – when was it observed? Most values in EPN-TAP come as pairs of min and max.
- `c<n>_min`, `c<n>_max` – where is it? Compared to core astronomy, solar system science is plagued by a plethora of coordinate systems. Hence, there is no RA and Dec, but rather three generic coordinate intervals. What they actually mean is given by `spatial_frame_type` (which could be something like "cylindrical"; in the solar system, you have a lot more than just the spherical coordinats that are fine for most of core astronomy) and some identifier for how to interpret these numbers `spatial_coordinate_description` (which would correspond to thing like ICRS or Galactic on the sky). You will need to constrain at least the latter if you expect any sensible result to come out of spatial constraints.
- `dataproduct_type` – the sort of observation. This is like the eponymous column in Obscore, except that these are hashlists of 2-letter codes at this point, defined in the standard itself (Erard and Cecconi et al., 2022) rather than in the product-type vocabulary.
- `instrument_host_name` – the probe or laboratory that produced the data. Again, at this point it is not certain what strings would match a given probe; here, however, there is hope that soon-ish a vocabulary will be produced.
- `instrument_name` – the instrument that produced the data. Again, you have to bascially guess what the instrument is called, and the column may contain a hashlist.

EPN-TAP 2: Hashlists

Many EPN-TAP fields are "hash lists": they are actually multivalued, and to still keep everything in one table, multiple values are concatenated by hashes (#), as in an instrument name like

Visible Infrared Thermal Imaging Spectrometer#VIRTIS

To match such columns, use the `ivo_hashlist_has(hashlist, item)` UDF.

EPN-TAP 3: Global Discovery

Global EPN-TAP discovery means: query all epncore tables. To find these, you have to:

- look for resources containing epncore tables at all and then
- find the tables implementing epncore in them.

To make things even more complicated, essentially all EPN-TAP tables appear twice: Once in a record dedicated to them (with author, title, description for the table itself), once in the TAP service that hosts them. We only want to match the first kind, which for technical reasons is done in pyVO by only accepting a resource record if it has an access mode `tap#aux`.

In code:

```
def iter_epncore_tables(*args, **kwargs):
  for resrec in pyvo.registry.search(datamodel="epntap", *args, **kwargs):
    if not 'tap#aux' in resrec.access_modes():
      continue

    for tab in resrec.get_tables().values():
      utype = tab.utype or ""
      if (utype=='ivo://vopdc.obspm/std/epncore#schema-2.0'
          or utype.startswith('ivo://ivoa.net/std/epntap#table-2.')):
        yield resrec, tab
```

Ψ epnquery.py

This will only work on pyVO later than 1.5, because in 1.5 the table utype was not exposed. In case you wonder what the `yield` statement does: it makes the function a *generator*. This means that you can iterate over its result without having to create a sequence in between.

The inner loop examines the tables published by the resource; tables conforming to EPN-TAP are identified by a utype, which is some characteristic string saying about as much as "something to do with data models". In this case, there are still two substantially different utypes around in the VO, one created during the development of the standard (the one with the vopdc.obspm authority), one for the final IVOA standard. Hence, we need to match against both for the time being. The ivoa.net identifier will also change as future (minor, i.e., compatible) versions of EPN-TAP come around, which is why we do a prefix match. This second constraint is what will be enough in a future when all the legacy services will be updated.

The entire extra function is necessary here because we are not only discovering full resources here (the normal "unit of discovery" in the VO Registry) but have to discover tables on top. I expect pyVO will grow a function that will isolate you from these technicalities in the future; it may be worth perusing the current documentation when you need to do something like this in practice.

Doing something with our results is a bit more complex here than in the, say, obscore case, because EPN-TAP lets people put almost any kind of data into such tables, and what your

`access_url` points to – spectra, profiles of elemental abundances, odd magnetospheric data, or nothing at all – is impossible to tell before at least inspecting the `dataproduct_type` column (and even then your average non-solar-system astronomer may be stumped...). Hence, in our example we restrict ourselves to simply send any non-empty result to TOPCAT.

In mid-2024, the program will also fail with a syntax error when it hits the VizieR EPN-TAP service, because the do not properly quote their table name; with a bit of luck, this problem will be gone by the time you read this.

Exercise 22

Get the epnquery.py and change it to only discover spectra (that's dataproduct type *sp* in EPN-TAP). then send the first two spectra your program finds to TOPCAT (or SPLAT, or CASSIS, if you have one of them).

8.2 Custom Parameters to Simple Services

Custom Parameters: Discovery

SIAP only has very few standard parameters (e.g., no time constraints), and even SSAP's rich parameter set is insufficient for, e.g., theoretical spectra.

SIAP and SSAP services can define custom parameters. Discover them using a `FORMAT=METADATA` URL parameter.

The input parameters are given as VOTable params in the root VOTable RESOURCE, where their names are prefixed with `INPUT:`. You can figure out names, units, descriptions, and, if the service operators do a good job, even hints as to what you should pass in when you want to get data back.

pyVO does not yet have some API that would properly hide this (not terribly pretty) implementation detail. Worse, it is not totally trivial to get these PARAMs with astronomer-level pyVO.

To make amends, this course comes with a script `viewparams.py` that has a function and a UI to retrieve metadata. To see how an example works, try

```
python viewparams.py "http://dc.g-vo.org/bgds/q/sia/siap.xml?"
```

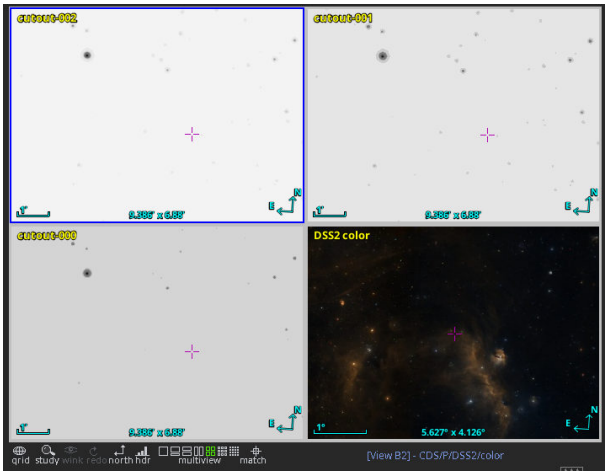
Ψ viewparams.py

Custom Parameters: Usage

Pass custom parameters as keyword arguments to search:

```
svc.search((107, -10), (0.05, 0.05),
  dateObs="57050/58050",
  bandpassId="SDSS i'")
```

Ψ siapextra.py



Custom Parameters: Syntax Trouble

We often have to pass intervals. You need some syntax to write upper/lower limits.

Old-style VO services (most of them) have intervals declared as `char[*]` or `double`) and expect `min/max`.

Others have two simple float parameters with `_MIN` and `_MAX`.

New-style (SIAv2, datalink...) services have *interval* xtypes and type `double[2]`. These intervals are written with a blank.

We are sorry about this, but not all standards work out well on the first attempt. In defence of the early standards authors that came up with the wretched slash syntax: There was prior un-art for this from the geospatial community.

Exercise 23

The SSAP service at http://dc.g-vo.org/theossa/q/ssa/ssap.xml? houses theoretical spectra mostly of hot, compact stars (think central stars of planetary nebula or perhaps young white dwarfs).

See if you can retrieve three spectra for stars with `log_g` between 4.5 and 5.5, an effective temperature between 7×10^4 and 10^5 Kelvin, and a Nitrogen mass fraction larger than 0.015 dex (write `+Inf` for "no upper limit").

Send the spectra retrieved to splat.

Hints: Use `viewparams.py`, start from `siapextra.py`, remember `dal.ssa.SSAService`, and pass in `FORMAT='VOTable'` to avoid retrieving spectra in both FITS and VOTable.

Use `pyvo.samp.send_spectrum_to`; this needs a URI of the spectrum, which you will find using the `getdataurl` method or what you get back from `search`. Note that current splat-s will not start a SAMP hub themselves, so you will need to start, for instance, TOPCAT first. Feel free to try another spectral client if you want.

You cannot directly use `send_spectrum_to` to send the spectra to TOPCAT, because TOPCAT does not subscribe to spectra. You *could*, however, make an astropy table out of the spectrum using its URL and then `send_table_to` as before.

8.3 TAP Uploads: The right way

Efficient Uploads: The Problem

TAP uploads are powerful, but they do have limits. In general, you cannot upload billion-row tables and expecte services to go along.

To make things fast and save the server's resources, you should only upload enough to select the relevant data. So, avoid:

```
first_result = svc1.run_sync(...).to_table()
second_result = svc2.run_sync(
  "SELECT * FROM local.t JOIN TAP_UPLOAD.up as b USING (foo, bar)",
  uploads={"up": first_result})
```

– this will upload all of `first_result` and download it right again; transferring data you already have, ingesting it into the remote database in between is just a waste of resources.

Efficient Uploads: The Pattern

Instead, if you want to join on first_result's columns foo and bar, make a new local table containing just those plus a unique local identifier (add a record number if no such identifier exists), somewhat like this:

```
first_result = svc1.run_sync(...).to_table()
remote_match = svc2.run_sync(
  "SELECT * FROM local.t JOIN TAP_UPLOAD.up as b USING (foo, bar)",
  uploads={"up": table.Table([
    first_result["main_id"],
    first_result["foo"],
    first_result["bar"])})
full_result = table.join(
  first_result,
  remote_match.to_table(),
  keys="main_id")
```

Efficient Uploads: Slicing

If you still run into resource limits, you process your data in batches. Use case: retrieve quality measures for Gaia DR3 data by matching on Gaia's `source_id`.

```
def iter_slices(total_length, batch_size):
    limits = list(range(0, total_length, batch_size))+[batch_size]
    for lower, upper in zip(limits[:-1], limits[1:]):
        if lower < upper:
            yield slice(lower, upper)


def remote_match(svc, source_table, remote_table, batch_size, match_column):
    matched_records = []
    match_on = source_table[match_column]

    # only match the match_column (for a positional crossmatch, use
    # an id column (create one if necessary) and the positions).
    for slice in iter_slices(len(source_table), batch_size):
        result = svc.run_sync(
            f"""SELECT a.* FROM
                {remote_table} AS a JOIN
```

```
            TAP_UPLOAD.mine AS b
            USING ({match_column})"""),
        uploads={"mine": table.Table([match_on[slice]])})
    matched_records.append(result.to_table())

joined_match = table.vstack(matched_records)
return table.join(source_table, joined_match, keys=match_column)
```

Ψ smart-tap-upload.py

This example is only *somewhat* contrived: For instance, in the result, you can compare the plain `ruwe` – which says how much you may trust Gaia's solution – with `fidelity_v2` – which says something similar, but *may* be a bit more meaningful, as it takes into account a source's environment –, and you can then look for systematics on, say, magnitudes or parallaxes.

Do not be alarmed by the MergeConflictWarnings; these are because the metadata of the `source_id` column different between the two TAP services participating (ESAC and GAVO) here.

Exercise 24

Add full Gaia records from ivo://esavo/gaia/tap's DR3 `gaia_source` to some records from the hdgaia.main table on GAVO's data centre. This does not need any slicing; still, only upload what you actually need for matching; for that, the smart-tap-upload.py example should be helpful.

Hint: for our simple table.join to work (which needs the same name in both tables), it is probably smart to rename `source_id3` in hdgaia at the ADQL level.

9 Solutions for Most of the Exercises

Solution for Exercise 1 I found an access URL by typing **ROSAT survey pointed** into a freetext constraint in WIRR and adding a *Service Type* constraint of *Image Access*. This, at the moment, only leaves one service, the SIA link of which I pasted into the program.

The resulting code is:

```
import pyvo

ACCESS_URL = "http://dc.zah.uni-heidelberg.de/rosat/q/im/siap.xml?"

svc = pyvo.sia.SIAService(ACCESS_URL)
images = svc.search((340.1,3.36), size=(0.1, 0.1))
image=images[0]
print(image.filesize, image.instr)
```

If you got an exception like

```
IndexError: index 0 is out of bounds for axis 0 with size 0
```

– this is an artefact of how we blindly fetch the first result. What this really means: there was no match in the service. Depending on what part of the ROSAT results is published, it is totally conceivable that they did not cover our location.

Solution for Exercise 2 To obtain the position of M51, I am using the SkyCoord snippet from the pyVO documentation; the rest mainly is cleanup and a standard random hack:

```
import random
import sys

from astropy import coordinates
from astropy.time import Time
from pyvo.dal import sia
from pyvo import registry

POS = coordinates.SkyCoord.from_name('M51')

def search_one_resource(res_rec):
    print("\nNow querying ", res_rec.res_title)
    svc = res_rec.get_service("sia", lax=True)
    images = svc.search(POS, size=0.5)
    for match in images:
        print(f"{match.title} Get? ", end=" ")
        if input().strip().lower().startswith("y"):
            match.cachedataset()

def main():
    for res_rec in registry.search(servicetype="image"):
        if random.random()<0.9:
            continue
        try:
            search_one_service(res_rec)
        except KeyboardInterrupt:
            if input("\nQuit? ").strip().lower().startswith("y"):
                sys.exit()
        except:
            import traceback
            traceback.print_exc()

if __name__ == "__main__":
    main()
```

If you are somewhat downtrodden by how much breakage you see and how weird some of the images that you find look like: Relax, it's science. There is actually a lot of art and knowledge between the raw images and the pretty pictures you see in the newspaper.

Solution for Exercise 3 The source code in question is in `pyvo/samp.py` (at the time of writing; it might be moved at some point).

The connection manager is right at the bottom of the file, and you see there that code like this should connect you to the SAMP hub:

```
client = SAMPIntegratedClient(name="test", description="VO course problem solution")
client.connect()
```

At least in the astropy versions current while this was written, when the client object just vanishes, it will not tell the hub the client is dead, and hence zombie clients will aggregate, for instance in TOPCAT's SAMP client line. Even if astropy were to get a bit smarter here, objects are in a precarious state when the automatic garbage collection strikes. Doing explicit connection management therefore is highly preferable in any case. In particular if it is as simple as just using a context manager.

Incidentally, on Debian-derived systems an attractive alternative to feeding github behavioural data would be to say `apt-get source python3-pyvo`.

Solution for Exercise 4 You will find that `send_image_to` calls `send_product_to`, just filling in the latter's `mtype` argument. More on the MTypes later; consider them a function name.

Now, `send_product_to` basically fills a dictionary like this:

```
message = {
        "samp.mtype": mtype,
        "samp.params": {
            "url": url,
            "name": name,
        },
    }
```

This is basically like a function call with keyword arguments. That really is almost all the magic; knowing this may come in useful if you want to send out more tailored SAMP messages later.

To send the image to Aladin only, you can guess that you will want to use the `client_name` argument (don't worry about the implementation in that case). In TOPCAT's SAMP status, you can find that Aladin's client name is a capitalised "Aladin", so in sum, you would say:

```
pyvo.samp.send_image_to(
   conn,
   match.acref,
   name=match.suggest_dataset_basename(),
   client_name="Aladin")
```

Solution for Exercise 5

```
import pyvo

svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
result = svc.run_sync("SELECT count(*) as ct FROM arihip.main")
print(result[0]["ct"])
```

Solution for Exercise 6 The error message will have looked a bit like this:

> Field query: Could not parse your query: Expected end of text, found '4' (at char 64), (line:3, col:17)

This is admittedly not terribly helpful, but it is surprisingly hard to get parsers – programs that turn sequences of characters into some structured representations – to spit out helpful error messages. When you do not understand what some error means, first look at the position the machine gave up at. In this case, this means showing the query that was actually executed; a good `print` is perfectly fine in these cases, but my advice is to drop into the debugger, which you can do with a line like this:

```
import pdb; pdb.set_trace()
```

You are then dropped into something you can interact with (try **help** at the prompt), e.g., by evaluating python expressions:

```
-> print(QUERY.format(**locals()))
(Pdb) QUERY.format(**locals())
"select accref, imagetitle\nfrom maidanak.reduced\nwhere object=IC 4756"
(Pdb) cont
```

If you are fluent in ADQL, you will notice that at the error position reported by the server, there is a naked number. And that is because of our extremely simple-minded templating: A "good" templating engine should have turned the python string into an ADQL string literal. But as I said, if you control both sides, it is fine to just adjust the template to:

```
QUERY = """select accref, imagetitle
from maidanak.reduced
where object='{object}'"""
```

Solution for Exercise 7 An adapted version of fetch3 would look like this:

```
import pyvo

QUERIES = [
  ("tgas", "http://dc.zah.uni-heidelberg.de/tap",
    """SELECT ra, dec, pmra, pmdec
        FROM tgas.main
        WHERE phot_g_mean_mag BETWEEN 8 AND 8.2"""),
  ("rave", "http://dc.zah.uni-heidelberg.de/tap",
    """SELECT raj2000, dej2000, rv, hmag
        FROM rave.main
        WHERE hmag BETWEEN 8 AND 8.2"""),]

def main():
  with pyvo.samp.connection() as conn:
    for short_name, access_url, query in QUERIES:
      service = pyvo.dal.TAPService(access_url)
      result = service.run_sync(query.format(**locals()), maxrec=90000)
      pyvo.samp.send_table_to(
        conn,
        result.to_table(),
        client_name="Aladin",
        name=short_name)


if __name__=="__main__":
  main()
```

As hinted, the secret of attractive plots in Aladin are filters. For TGAS, you can use the predefined "Draw proper motions of stars" filter and perhaps improve it a bit in the "Advanced Mode", e.g., by multiplying the two columns with 10.

For the radial velocity, perhaps a rainbow (blue and redshift) is appropriate? With a bit of experimentation I have come up with

```
{ draw rainbow(${rv}, -100, 100) fillcircle(300)}
```

Solution for Exercise 8 The two workarounds are:

- `work_around_vizast_bug` – this used to be necessary because VizieR used to put `arraysize="1"` into column declarations of their scalars, which made astropy make arrays from them. This has long been fixed.
- `work_around_sdss_ucd_bug` – this used to be necessary because the UCDs on the SDSS table were wrong and non-specific at the same time. If you drop the workaround, perhaps by writing `return ucd` at the top of the function, you will see that the optical part of our SEDs will be gone; the UCDs are still too unspecific for our purpose. Once you have understood what happens in the workaround, you can also use TOPCAT's table browser to ascertain that the UCDs are still only *phot.mag;em.opt*.

Solution for Exercise 10 Here is how to write this:

```
import pyvo

QUERIES = {
  "tgas": ("http://dc.zah.uni-heidelberg.de/tap",
    """SELECT *
        FROM
          tgas.main AS tg
          JOIN TAP_UPLOAD.rave AS mine
          ON DISTANCE(tg.ra, tg.dec, mine.raj2000, mine.dej2000)<1/3600.
    """),
  "rave": ("http://dc.zah.uni-heidelberg.de/tap",
    """SELECT raj2000, dej2000, rv, hmag
        FROM rave.main
        WHERE hmag BETWEEN 8 AND 8.1"""),}


def main():
  svc_url, query = QUERIES["rave"]
  rave_svc = pyvo.dal.TAPService(svc_url)
  job = rave_svc.submit_job(query, maxrec=90000)
  try:
    job.run()
    job.wait()
    job.raise_if_error()

    svc_url, query = QUERIES["tgas"]
    tgas_svc = pyvo.dal.TAPService(svc_url)
    result = tgas_svc.run_sync(query,
      uploads={
        "rave":  job.result_uri})

  finally:
    job.delete()

  with pyvo.samp.connection() as conn:
    pyvo.samp.send_table_to(
      conn, result.to_table(), client_name="topcat", name="rave+tgas")


if __name__=="__main__":
  main()
```

Note how this is much more logical than the first version with the individual photometric cuts, since there is just one constraint on the magnitudes now (the one on the H-band in rave)

– when you send the resulting table to Aladin, you will see more matches in TGAS than you had when you were comparing the two catalogue cuts manually.

And no, I would not normally have kept queries and access URLs in a dictionary in a situation like this; the two queries are different roles, and representing them next to each other is misleading rather than helpful. I wrote it like this in order to keep the program structure as parallel to the original rave-tgas solution as possible.

If you inlined the queries, you actually showed better taste.

By the way, you could take this even further and make the tgas query async as well. You could then send a raw *table.load.votable* message to TOPCAT with the result URL as the table URL. That way, pyVO would not touch the data at all. That is a small win here, but it might be a useful technique in more demanding circumstances.

To do that, your send query would look like this:

```
svc = pyvo.dal.TAPService("http://dc.zah.uni-heidelberg.de/tap")
job2 = svc.submit_job("""
    SELECT *
    FROM tgas.main AS db
    JOIN TAP_UPLOAD.t1 AS tc
      ON DISTANCE(db.ra, db.dec, tc.raj2000, tc.dej2000) < 5./3600.
    """,
    uploads={"t1": job.result_uri})
job2.run()
job2.wait()
message = {
    "samp.mtype": "table.load.votable",
    "samp.params": {
        "url": job2.result_uri,
        "name": "tgas+rave-from-server",
    },
}
client_id = samp.find_client_id(conn, "topcat")
conn.call_and_wait(client_id, message, "10")
```

The main complication over the code above is that we `send_table_to` cannot deal with remote URIs and we have to essentially copy its implementation. But this is still relatively compact, I would say.

Solution for Exercise 11 Just add a `AND em_res_power>10000` to the query in the program.

And it is totally conceivable that you will not find anything for the objects you chose. Spectra of this sort are expensive to get and have only been obtained for relatively few stars

Solution for Exercise 12 To answer this, you could of course read the documentation and figure out what's the name of the respective properties of the TAPService object. In fact, you should set aside an hour or two to at least browse the documentation of pyVO if you use it regularly (as you should with any other library that you regularly use). However, in this case discovery with command line completion in (i)python is legal; for instance,

```
In [3]: svc = pyvo.dal.tap.TAPService("http://dc.g-vo.org/tap")

In [4]: svc.<tab>
  svc.availability   svc.create_query   svc.run_async      svc.tables
  svc.available      svc.describe       svc.run_sync       svc.up_since
  svc.baseurl        svc.hardlimit      svc.search         svc.upload_methods
  svc.capabilities   svc.maxrec         svc.submit_job
```

It is a reasonable guess that `maxrec` and `hardlimit` are what you are looking for in this case. A program doing what is asked for in the exercise would look somewhat like this:

```
from pyvo import registry

for res_rec in registry.search(keywords="tgas", servicetype="tap"):
  svc = res_rec.get_service("tap")
  print(svc.baseurl, svc.maxrec, svc.hardlimit)
```

Solution for Exercise 13

```
>>> from pyvo import registry
>>> rscs = registry.search(keywords="exoplanet merged catalogue")
>>> rscs.get_summary()
(we got it, short name ExoMerCat)
>>> svc = rscs["ExoMerCat"].get_service("tap")
>>> list(svc.tables.keys())
['exomercat.exomercat', ...
>>> svc.tables["exomercat.exomercat"].columns
[... <BaseParam name="ra_off"/>, <BaseParam name="dec_off"/>, <BaseParam
name="mass"/>...]
>>> res = svc.run_sync("select top 1 ra_off, dec_off from exomercat.exomercat order by mass asc")
>>> res[0]
(277.6981916666667, -10.991083333333332)
>>> cres = registry.search("constellation polygons")
>>> cres.get_summary()
<Table length=1>
index short_name ...        interfaces
int32    str9    ...          str24
----- ---------- ... ------------------------
    0  cstl cone ... conesearch, tap#aux, web
>>> csvc = cres[0].get_service("tap")
>>> csvc.tables["cstl.geo"].columns
[..., <BaseParam name="name"/>, <BaseParam name="p"/>, <BaseParam name="ra"/>, ...]
>>> csvc.run_sync("select name from cstl.geo as db join tap_upload.pt as mine"
...     " on 1=contains(point(mine.ra_off, mine.dec_off), p)",
...     uploads={"pt": res.to_table()}).to_table()
<Table length=1>
 name
object
------
Scutum
```

So: with sufficient instrumentation and a clear horizon, you could see its host star from here (as in: Heidelberg). Scutum is just a bit south of the celestial equator, visible between Atair and Antares in summer nights.

Note that, of course, this is exactly not the nice blind (i.e., without prior knowledge of concrete resources) discovery we would like to have in the VO. But bear with us: It's much easier to write problems assuming prior knowledge.

Solution for Exercise 14 For server-side-expansion, change the condition to use the `gavo_vocmatch` UDF mentioned in the side track, like this:

```
    def __init__(self, uat_id, expand=False):
        if expand:
            self._condition = "1=gavo_vocmatch('uat', {uat_id}, uat_concept)"
```

```
        else:
            self._condition = "{uat_id} = uat_concept"
        self._fillers = {"uat_id": uat_id}
```

If you try it, you will notice that you get back massively more services.

When doing things locally, there is a complication because the naive templating engine cannot cope with sets. If it could, you would be done with just pulling the vocabulary (in a class attribute so we don't parse the vocabulary each time we are called) and then using a different operator:

```
class ForSource(pyvo.registry.SubqueriedConstraint):
    _keyword = "subject"
    _subquery_table = "rr.subject_uat"

    uat_voc = pyvo.utils.vocabularies.get_vocabulary("uat")

    def __init__(self, uat_id, expand=False):
        if expand:
            uat_ids = {uat_id} | set(self.uat_voc["terms"][uat_id]["narrower"])
        else:
            uat_ids = {uat_id}

        self._condition = "uat_concept in ({uat_ids})".
        self._fillers = {"uat_ids": uat_ids}
```

As things are, this will lead to an error, because the templating engine has no idea what to do with your set. Hence, you will have to manually do your formatting. But note that this sort of hack will make you vulnerable to SQL injection, so *never* create SQL like this when processing untrusted content:

```
        uat_ids = {uat_id} | set(self.uat_voc["terms"][uat_id]["narrower"])
        self._condition = "uat_concept in ({})".format(
            ", ".join(f"'{id}'" for id in uat_ids))
```

Solution for Exercise 15

```
def get_available_semantics(dl):
  res = set()
  for link in dl:
    res.add(link["semantics"])
  return res
```

Solution for Exercise 16 The first hurdle to the solution regrettably is: how will that service spell the identifier "IC 1151"? Using TOPCAT's or pyVO's table browsers, you will find the `target_name` column, and using something plausible like

```
select target_name from califadr3.cubes where target_name like '%1151%'
```

you will find they have skipped the blank (oh, for interoperable object designations!), and hence you will have to match IC1151.

For the matter with the setup, you can guess that it's what the setup column says and you would end up constraining `setup` to COMB (which in this case says that you are using a clever combination of the results of a higher and a lower resultion setup).

With these preparations, you can do the SODA calls; because, at the time for writing, pyVO does not pick up the processing service sitting on the dataset (rather than result set) level, we need to do the extra complication handled in the `get_cutout_frame` function given in the problem statement.

Taking everything together:

```
import io
import pyvo
import numpy
from astropy import units as u
from astropy.io import fits
from PIL import Image

def _normalize_for_image(pixels):
  pixels = numpy.flipud(pixels)
  pixMax, pixMin = numpy.max(pixels), numpy.min(pixels)
  pixels = (pixels-pixMin)/(pixMax-pixMin)*255
  return numpy.asarray(pixels, numpy.uint8)


def get_cutout_frame(datalink, wavelength):
  proc = datalink.get_first_proc()
  fits_stream = proc.processed(band=(wavelength, wavelength))
  return fits.open(io.BytesIO(fits_stream.read()))[0].data[0]


svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
res = svc.run_sync(
  """SELECT * FROM califadr3.cubes
  WHERE target_name='IC1151' AND setup='COMB'""")
datalink = next(res.iter_datalinks())
pixels = numpy.array([
  _normalize_for_image(get_cutout_frame(datalink, 700*u.nm)),
  _normalize_for_image(get_cutout_frame(datalink, 550*u.nm)),
  _normalize_for_image(get_cutout_frame(datalink, 400*u.nm))])
pixels = pixels.transpose(1,2,0)

Image.fromarray(pixels, mode="RGB"
  ).save("IC1151.jpeg", format="jpeg")
```

if you cannot get enough: It is not hard to extend this so for each band, a few spectral frames are averaged rather than just exactly one frame (which we pick out here with our relatively stupid `.data[0]`.

Solution for Exercise 17 You would probably replace the service creation in the constructor's class with

```
self.sia_service = pyvo.dal.SIAService(
  "http://dc.g-vo.org/lswscans/res/positions/siap/siap.xml")
```

With that, `make_response_table` would be as simple as:

```
ra = self.cur_table[self.ra_name][table_index]
dec = self.cur_table[self.dec_name][table_index]
return self.sia_service.search(pos=(ra, dec), size=0.05).to_table()
```

Solution for Exercise 18

```
import pyvo
from astropy.coordinates import SkyCoord
from astropy import units as u

import vohelper

class Odometer:
  def __init__(self, conn):
    self.conn = conn
    self.total_travelled = 0*u.deg
    self.last_position = None
    conn.bind_receive_message("coord.pointAt.sky", self.record_movement)

  @vohelper.show_exception
  def record_movement(self, privkey, sender_id, msg_id, mtype, params, extra):
    new_coord = SkyCoord(
      float(params["ra"])*u.deg,
      float(params["dec"])*u.deg)

    if self.last_position is not None:
      self.total_travelled += new_coord.separation(self.last_position)

    self.last_position = new_coord
    print(self.total_travelled)


def main():
  with pyvo.samp.connection(addr="localhost") as conn:
    odometer = Odometer(conn)
    input()
  print("Total travelled", odometer.total_travelled)


if __name__=="__main__":
  main()
```

Solution for Exercise 23

```
import pyvo

svc = pyvo.ssa.SSAService("http://dc.g-vo.org/theossa/q/ssa/ssap.xml?")
with pyvo.samp.connection() as conn:
  for ct, result in enumerate(
      svc.search(pos=None, diameter=None, t_eff="70000/100000",
        log_g="4.5/5.5", w_N="0.015/+Inf",
        FORMAT="VOTable")):

    pyvo.samp.send_spectrum_to(conn, result.getdataurl(), client_name="splat")
    if ct==2:
      break
```

Solution for Exercise 24

```
import pyvo
from astropy import table

gavo_svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
hd_subset = gavo_svc.run_sync("SELECT TOP 500 mv_meas, spectral,"
```

```
    " source_id3 as source_id FROM hdgaia.main").to_table()

esac_svc = pyvo.dal.TAPService("https://gea.esac.esa.int/tap-server/tap")
gaia_meta = esac_svc.run_sync(
    """SELECT *
        FROM gaiadr3.gaia_source as g
        JOIN tap_upload.mine as m
        USING (source_id)""",
    uploads={"mine": table.Table(
        [hd_subset["source_id"]])})
full_result = table.join(
    hd_subset,
    gaia_meta.to_table(),
    keys="source_id")
with pyvo.samp.connection() as conn:
    pyvo.samp.send_table_to(conn, full_result, client_name="topcat")
```

References

Demleitner, M., Dowler, P., Plante, R., Rixon, G. and Taylor, M. (2012), 'TAPRegExt: a VOResource Schema Extension for Describing TAP Services Version 1.0', IVOA Recommendation 27 August 2012, arXiv:1402.4742. doi:10.5479/ADS/bib/2012ivoa.spec.0827D, https://ui.adsabs.harvard.edu/abs/2012ivoa.spec.0827D.

Demleitner, M., Harrison, P., Molinaro, M., Greene, G., Dower, T. and Perdikeas, M. (2019), 'IVOA Registry Relational Schema Version 1.1', IVOA Recommendation 11 October 2019. doi:10.5479/ADS/bib/2019ivoa.spec.1011D, https://ui.adsabs.harvard.edu/abs/2019ivoa.spec.1011D.

Erard, S., Cecconi, B., Le Sidaner, P., Demleitner, M. and Taylor, M. (2022), 'EPN-TAP: Publishing Solar System Data to the Virtual Observatory Version 2.0', IVOA Recommendation 22 August 2022. https://ui.adsabs.harvard.edu/abs/2022ivoa.spec.0822E.

Mantelet, G., Morris, D., Demleitner, M., Dowler, P., Lusted, J., Nieto-Santisteban, M. A., Ohishi, M., O'Mullane, W., Ortiz, I., Osuna, P., Shirasaki, Y. and Szalay, A. (2023), 'Astronomical Data Query Language Version 2.1', IVOA Recommendation 15 December 2023. https://ui.adsabs.harvard.edu/abs/2023ivoa.spec.1215M.

This document's DOI is 10.21938/08rzo4ylRPmnS8iXYPO:rg.