A Short Course on pyVO

Markus Demleitner Hendrik Heinl

August 22, 2024

German Astrophysical Virtual Observatory

Introduction

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

pyVO Basics

Prerequisites

- python and astropy, of course
- TOPCAT for viewing and visualising tables
- Aladin 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`

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.

Ψ tap-obscore.ipynb

To fit things on slides, I am PEP 8-relaxed.

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...
```

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

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.

Also: `row.cachedataset` saves the image to your local disk under a name sensible for the metadata.

This is Python

The advantage of doing this in Python is that it is easy to add your own logic:

```
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

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; 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
```

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:
```

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.

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.

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"])
```

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

Step 2: Go Async

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

```
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

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()
```

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

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

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:

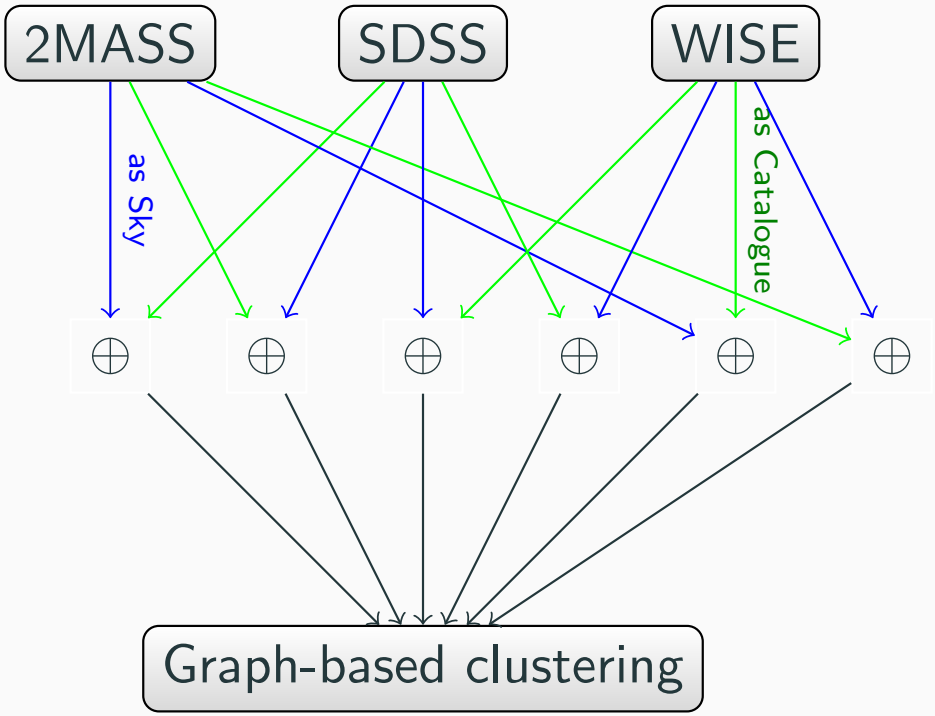
```
UCD_TO_WL = {
  "phot.mag;em.opt.u": 3.5e-7,
  "phot.mag;em.opt.b": 4.5e-7,
  "phot.mag;em.opt.v": 5.5e-7,
  "phot.mag;em.opt.r": 6.75e-7, ...}

  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))
```

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.



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

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?

Plan:

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

Obscore

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

- dataproduct_type – states *image*, *timeseries*, and the like.
- obs_publisher_did – a dataset identifier.
- 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.

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.

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})
```

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()
```

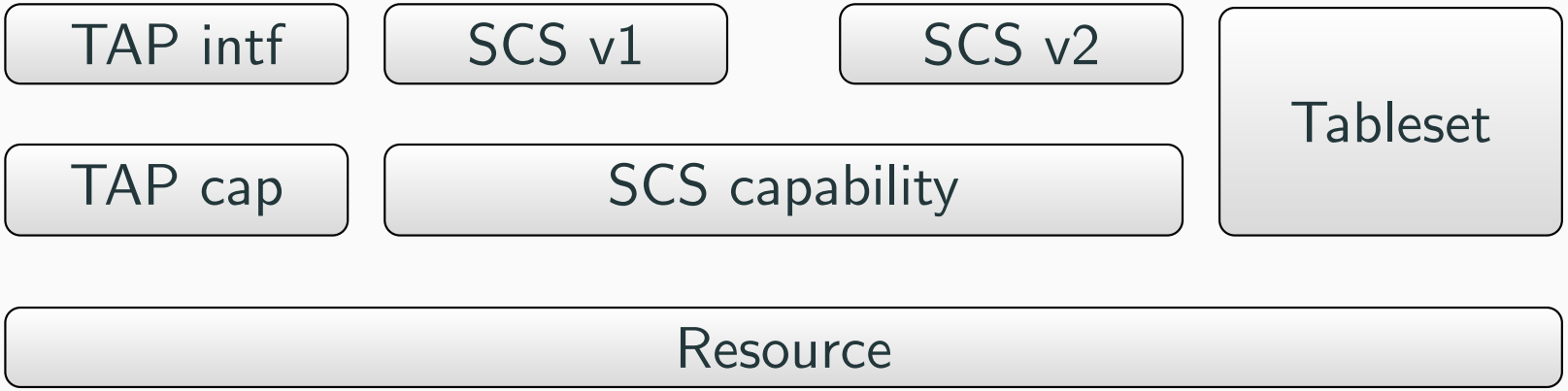
Ψ get-spectra.py

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:



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

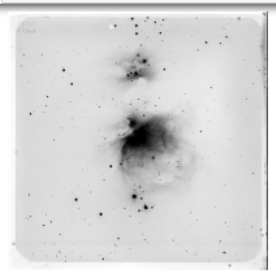
Interactive Use of the PyVO Registry API

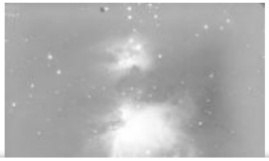
Finally: A jupyter notebook!

Ψ data-discovery-demo.ipynb

In order to have at least a few images in this notebook, let's use datalink to fetch a few previews of our matches (this datalink trick doesn't work on all services; if it doesn't complain to the operators, demanding datalink support – see the thing with get_contact above).

In [10]:
```
from IPython.display import Image, display
for dl in matches.iter_datalinks():
    for row in dl.bysemantics("#preview"):
        display(Image(url=row["access_url"], width=200,
                      embed=True, format="jpeg"))
```





There are similar constraints for the Spectral and Time axes. For instance, to look for resources talking about spectra and the Balmer break, you could say:

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.

To resolve an ivoid:

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

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

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.

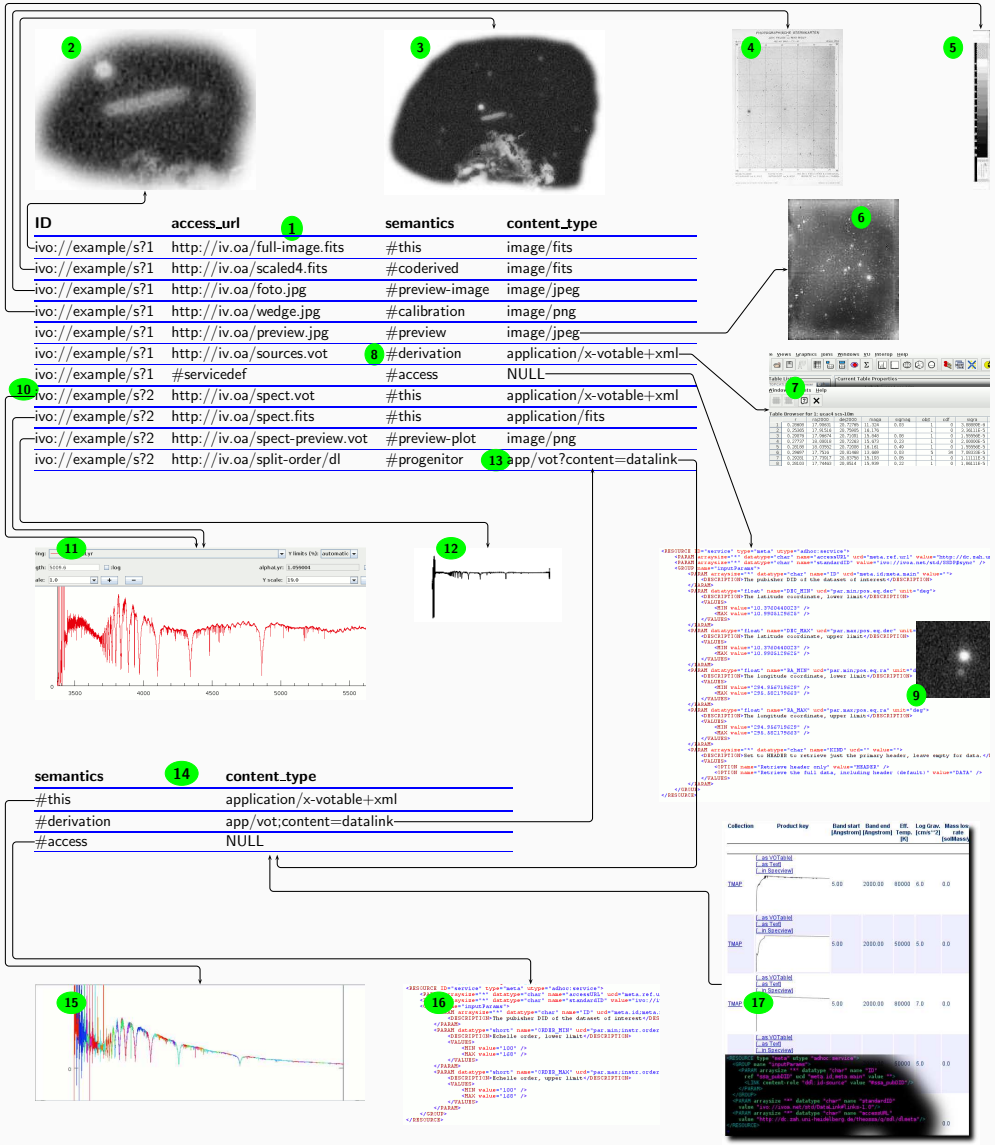
This is really machine-readable data; load any of these links into TOPCAT to inspect it as a VOTable:

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

Datalink in a Cartoon



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.

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

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

Datalink: Remote Processing on Non-Datalink Documents

Use case: Hα maps of Sd galaxies from CALIFA.

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`?

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

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.

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()
```

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):
    ...
```

Class name

Class variables

Constructor

Instance variables

Conventional self

Method definition

Ψ vicinitysearcher.py

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": {}})
```

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)
```

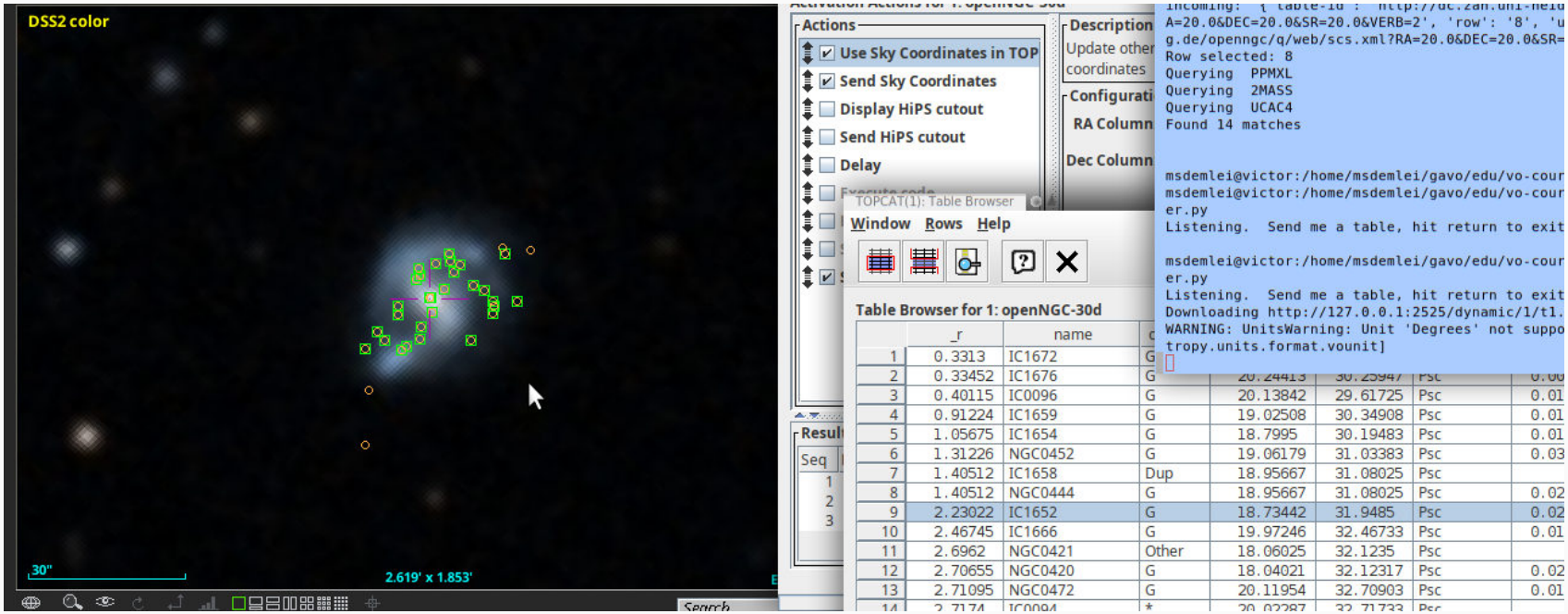
Try It Out

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.



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.

What *is* marginally possible: "Give me all measurements of a certain sort of UCD in a certain vicinity."

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)
```

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()
```

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),]
```

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,...)
```

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

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.

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.

Does a service support union?

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

Running Queries II: Adapting to Server Capabilities

```
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())
```

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!

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))
```

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")
```

Odds and Ends

EPN-TAP 1

EPN-TAP is like obscore, just for solar system data. Columns of note include:

- `granule_uid` – an identifier for the dataset
- `target_name` – what was observed?
- `time_min`, `time_max` – when was it observed?
- `c<n>_min`, `c<n>_max` – where is it?
- `dataproduct_type` – the sort of observation.
- `instrument_host_name` – the probe or laboratory that produced the data.
- `instrument_name` – the instrument that produced the data.

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.

```
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

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.

pyVO does not yet have some API that would properly hide this (not terribly pretty) implementation detail.

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

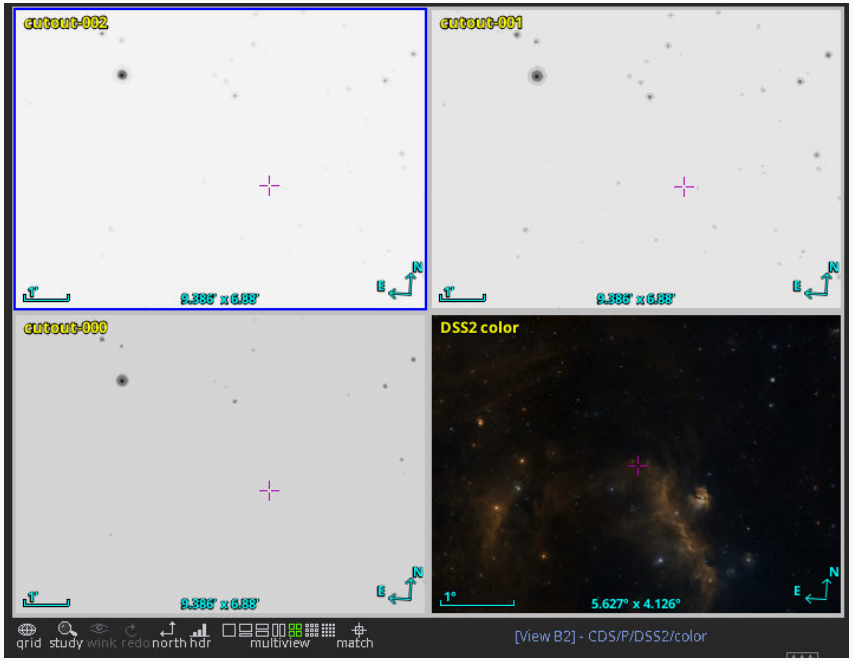
Ψ viewparams.py

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.

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
```