PyDA: The future of device access with Python

Phil Elson (BE-CSS-DSB)

3rd February 2025

CERN

Background: Python in ATS

- **There is a long history of Python use in the sector**
- **Python officially supported for operational controls 2019: Acc-Py**
  - Somewhat late to the Python party
  - Since then, have fully embraced the language
    - Acc-Py has been a success by many measures
    - Today we see that most users want to be using Python instead of Java
    - Wherever it makes sense, BE-CSS are gradually working towards decoupling our high-level controls libraries (clients) from Java

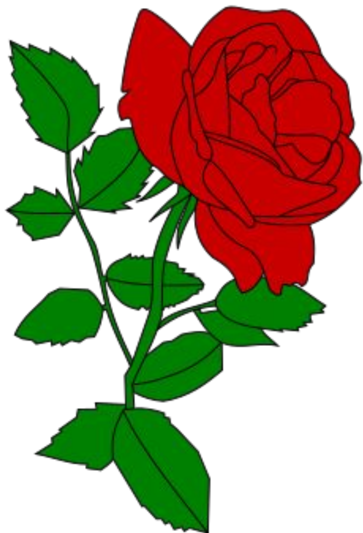
PyJapc

- **Created early 2016 (Tom Levens, Michael Betz, et al.)**
    - Built upon the Java API for Parameter Control (JAPC) - requires a JVM and all the controls Java libraries to work (quite some complexity!)
- **Maintained by BE-CO/BE-CSS since 2019**
- **Widely used for operational applications, expert tools, hardware R&D, MDs, etc.**
    - Some compelling features: it is easy to setup, has a simple interface, …
    - Some quirks*

- **<u>Breaking changes needed</u> in order to meet all user needs, iron out the quirks, and position it such that it could run without Java in the future**

* e.g. callback interface different between get/subscribe when getHeader=True/False; arrays of length 1 are converted to scalars; stateful client; interaction with pj global state;

~~PyJapc v3~~ ➜ PyDA



"What's in a name?
That which we call a rose by any other name would smell as sweet."

- ★ Ability install side-by-side with PyJapc
- ★ Introduce a data model which is forward looking and applies lessons learned from JAPC
- ★ A chance to address some of the fundamental issues (e.g. fixing the callback signature inconsistency)

PyDA development approach

Goal: To arrive at a design which meets the needs of the users, based on direct *feedback* from users.

- **Steady and intentional**
  - Not a full-time activity. Means we can gather more user input, and adapt as necessary
  - We started doing analysis on PyJapc, and the necessary breaking changes ~4 years ago
  - PyDA prototype first introduced ~3 years ago https://indico.cern.ch/event/974795
  - Numerous community updates and iterative releases since then
  - “PyDA hands-on experience in OP” presentation in Nov ‘24
  - “Fixing things” as we go (to CCDA, JAPC, RDA3, device metadata, JPype, …)

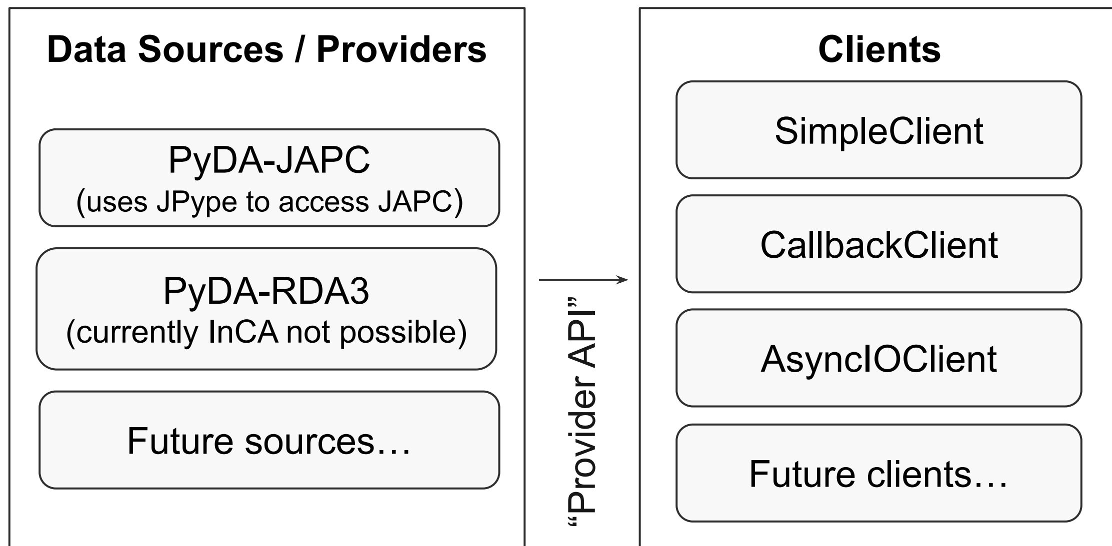
Expecting a PyDA v1 this month

PyJapc reaches end-of-life at the end of Run 3

PyDA scope and needs

- **<u>Online</u> device access - not a general device-data analysis framework**
  - Includes conceptual ability to replay recorded and simulated data within the Python process
- **A basis for future applications and services to be built upon**
  - It provides a common abstraction layer, meaning that many applications could choose to not build their own
- **Prioritising API expressiveness and avoiding unnecessary statefulness**
- **One size does not fit all: device access needs are different for interactive exploration** vs **online analysis** vs **those of a generic GUI**
  - Scope for new and hybrid clients in the future as needed (e.g. Qt signal-slot, Jupyter notebook specific, PyJapcScout, nicejapc, etc.)

PyDA high-level design



Notes

- Not tightly bound to JAPC
- Still able to leverage the capabilities of JAPC today (InCA/LSA for operational settings management)
- Able to adapt as new client needs arise
- Able to introduce new data sources (e.g. a data replay source)
- To avoid statefulness: Requires that the provider is passed to the client when created (in JAPC this is global state)

PyDA: Getting started

```
import pyda
import pyda_japc

provider = pyda_japc.JapcProvider()
client = pyda.SimpleClient(provider=provider)

response = client.get(
    'SOME.DEVICE/SomeProperty',
    context='SOME.TIMING.USER',
)
```

PyDA: Provider creation

```
import pyda
import pyda_japc

provider = pyda_japc.JapcProvider()
client = pyda.SimpleClient(provider=provider)

response = client.get(
    'SOME.DEVICE/SomeProperty',
    context='SOME.TIMING.USER',
)
```

PyDA: Instantiating a client

```python
import pyda
import pyda_japc

provider = pyda_japc.JapcProvider()
client = pyda.SimpleClient(provider=provider)

response = client.get(
    'SOME.DEVICE/SomeProperty',
    context='SOME.TIMING.USER',
)
```

PyDA: Requesting device data

```python
import pyda
import pyda_japc

provider = pyda_japc.JapcProvider()
client = pyda.SimpleClient(provider=provider)

response = client.get(
    'SOME.DEVICE/SomeProperty',
    context='SOME.TIMING.USER',
)
```

PyDA: The response object

```
response.
  p value                      PropertyRetrievalResponse
  p notification_type          PropertyRetri…
  p query                      PropertyRetrievalResponse
  p header                     PropertyRetrievalResponse
  p exception                  PropertyRetrievalResp…
  m __getattribute__(self, _…  object
  m __eq__                     PropertyRetrievalResponse
  m __hash__                   PropertyRetrievalRespo…
```

No matter which client, the response object type is the same. Details of the type in the PyDA documentation.

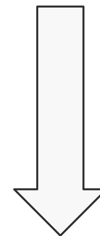
PyDA: Provider vs client

```
import pyda
import pyda_japc

provider = pyda_japc.JapcProvider()
client = pyda.SimpleClient(provider=provider)

response = client.get(
    'SOME.DEVICE/SomeProperty',
    context='SOME.TIMING.USER',
)
```

TIP: In general it is a lightweight operation to create a client. Creating a provider is more expensive.



If you ever wrote a function/interface which accepted a PyJapc instance, it can be replaced to accept a provider, leaving your implementation to instantiate the most useful client for your purpose.

PyDA: Using the callback client

```
import pyda
import pyda_japc

provider = pyda_japc.JapcProvider()
client = pyda.SimpleClient(provider=provider)

response = client.get(
    'SOME.DEVICE/SomeProperty',
    context='SOME.TIMING.USER',
)
```

PyDA: Using the callback client

```
import pyda
import pyda_japc

provider = pyda_japc.JapcProvider()
callback_client = pyda.CallbackClient( provider=provider)

def summarise_response (response: pyda.access.PropertyRetrievalResponse ):
    print(f'Property: {response.query.endpoint.property_name }')
    print(response.value)

callback_client.get(
    'SOME.DEVICE/SomeProperty' ,
    context='SOME.TIMING.USER' ,
    callback=summarise_response,
)
```

The signature of the callback for get, set, and subscribe is the same: a function which accepts a single argument (the response object).

PyDA: Using the callback client

```
import pyda
import pyda_japc

provider = pyda_japc.JapcProvider()
callback_client = pyda.CallbackClient( provider=provider)

def summarise_response (response: pyda.access.PropertyRetrievalResponse ):
    print(f'Property: {response.query.endpoint.property_name }')
    print(response.value)

subscription = callback_client.subscribe(
    'SOME.DEVICE/SomeProperty' ,
    context='SOME.TIMING.USER' ,
    callback=summarise_response,
)
subs.start()
```

Subscriptions run until the subscription handle is destroyed

PyDA and device metadata

- **Just like JAPC, PyDA understands more about a device than just the raw data it receives (e.g. via RDA3)**

- **The metadata source is part of the provider API**
  - By default, metadata is taken from the Controls Configuration Service (via PyCCDA)
  - In the future, it will also be able to take this directly from devices (FESA)

- **It means we can expose richer data objects in the response** (e.g. enumerations)**, and can safely interpret user input** (e.g. supporting Python integers when setting values)
  - We also expose the raw values, in case there is a need (e.g. for micro optimising an app, or device expert manipulation)

PyDA: Next steps

- **Expecting to release PyDA v1 in the coming days**
  - There is still work to do post-v1, but we don't think there will need to be further breaking changes
- **PyJAPC becomes end-of-life at the end of the run (mid-2026)**
  - We will issue a PyJapc release this year which *deprecates* the library
- **Explore where to go with tools built *on top* of PyJapc** **(e.g. PyJapcScout, nicejapc)**
  - Some parts may no longer make sense
  - Others we may wish to extract into core libraries (e.g. PyDA, event building library)
  - Provide (more) documentation and consultancy to help migrate from PyJapc to PyDA
    - Note: We already have a migration guide in the PyDA docs

MDs and Python: The perfect combination?

- **So far, PyDA has not had *much* feedback from the MD community**
  - Thanks to Simon Albright for suggesting this talk!
- **We would love to hear from you (on PyDA, and more broadly on the Acc-Py platform):**
  - What works well?
  - What could improve your workflow during MDs?
- **Starter topics:**
  - PyDA: Is there a need for notebook specialisations for online device data access?
  - PyDA: What are your needs for online data aggregation and synchronisation? (aka. event building, data table pivoting). Unlike PyJAPC, PyDA does *not* currently support parameter groups
  - Do Python virtual environments serve the MD community well?
  - Would access to the TN from SWAN be of interest to you? (work in progress: IT leading a project)

Thank you!

Find out more about PyDA:

https://acc-py.web.cern.ch/gitlab/acc-co/devops/python/prototypes/pyda/

or

Search for “pyda” at https://acc-py-repo.cern.ch/

Join us at the Acc-Py community meetings:

acc-python-announce eGroup

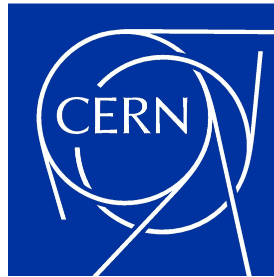
and

Mattermost

Try PyDA and let us know how it works for you.

Integrating your feedback into the PyDA is how it becomes an even more useful tool

Questions?



home.cern

Online vs offline data