Self-describing Portable Dataset Container

Maohai Huang[1,2,3]
mhuang@nao.cas.cn

[1] *National Astronomical Observatories, Chinese Academy of Sciences, Beijing, 100014, China*
[2] *University of Chinese Academy of Sciences, Beijing, China*
[3] *Key Laboratory of Space Astronomy and Technology, NAOC, Beijing, China*

Abstract

With SPDC one can pack data of different formats into modular dataset and Products, together with annotation (description and units) and metadata (Parameters about data). SPDC accommodates highly complex associated and nested structures.

Access APIs of the components of "SPDCs" are convenient, making it easier for scripting and data mining directly "on SPDCs". The toString() method of major container classes outputs nicely formatted text representation of complex data.

SPDCs are portable (de/serializable) in human-friendly standard format (JSON implemented), so that machine data processors on different platforms can parse, access internal components, or re-construct an SPDC. Even a human with a web browser can understand the data.

Most SPDC Products and components implement event sender and listener interfaces to facilitate scalable data-driven processing pipelines.

SPDC storage "pools" are provided for 1) data storage and, 2) for all persistent data to be referenced to with URNs (Universal Resource Names).

References of SPDC can become components of Context products, enabling SPDCs to encapsulate rich, deep, sophisticated, and accessible contextual data, yet remain light-weight.

For data processors, a web server with RESTful APIs is implemented, suitable for Docker containers in pipelines that mix legacy software or software of incompatible environments to form an integral data processing pipeline.

Introduction

SPDC is a ``container'' package written in Python for packing different types of data together, letting the container take care of inter-platform compatibility, serialisation, persistence, and data object referencing that enables lazy-loading. The word ``container'' in the name is more closely associated that in ``a shipping container'' instead of ``a Docker container".

One can associate groups, arrays, or tables of Products using basic data structures such as sets, sequences (Python `list`), mappings (Python `dict`), or custom-made classes that inherit functionalities from base classes provided by the package.

The following describes SPDC Python packages:

- The base data model is defined in package `dataset`.
- Persistent data access, referencing, and Universal Resource Names are defined in package `pal`.
- A reference REST API server designed to communicate with a data processing docker using the data model is in package `pns`.

dataset: Model for Data Container

SPDC aims to give pipeline data artifacts (products, intermedia data sets, auxiliary data sets) these properties:

Annotatable: one can use textual description to annotate the contained data;
Attributable: one can add attributes (or called properties, meta data) to the contained data;
Copyable: one can ask for a copy of the data;
Comparable: one can compare two containers to see if they are equal;
Queryable: Can be queried to discover its contents, and obtain references of the components
Serializable: one can transmit the data across the network and re-construct (deserialize) them on the receiving side
Accepts **Change Listeners**
Easy to handle with **RESTful API**

SPDC allows one to organize data into:
A **dataset** -- an arbitrary combination of
- N dimensional arrays with an optional unit,
- Tables
- a list of meta data.

Metadata – a list of named parameters
A **Parameter** – a string or quantity.
A **Product** -- an arbitrary combination of datasets with some mandatory meta data.

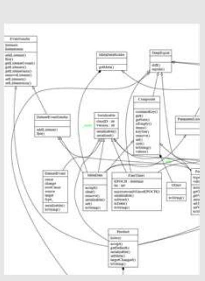
Python programming API examples for Dataset module:

Product

Creation
```
>>> x = Product(description="product example with several datasets",
...             instrument="Crystal-Ball", modelName="Mk II")
>>>
>>> print(x.meta['description'])  # == "product example with several datasets"
product example with several datasets
>>>
>>> print(x.instrument)  # == "Crystal-Ball"
Crystal-Ball
```

ways to add datasets
```
>>> i0 = 6
>>> i1 = [[1, 2, 3], [4, 5, i0], [7, 8, 9]]
>>> i2 = 'ev'                 # unit
>>> i3 = 'image1'  # description
>>> image = ArrayDataset(data=i1, unit=i2, description=i3)
>>>
>>> x["RawImage"] = image
>>> print(x["RawImage"].data)  # [1][2] == i0
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
>>> # no unit or description. different syntax but same function as above
... x.set('QualityImage', ArrayDataset(
...     [[0.1, 0.5, 0.7], [4e3, 6e7, 8], [-2, 0, 3.1]]))
>>> print(x["QualityImage"].unit)  # is None
None
>>> # add a tabledataset
... s1 = [('col1', [1, 4.4, 5.4E3], 'eV'),
...       ('col2', [0, 43.2, 2E3], 'cnt')]
>>> x["Spectrum"] = TableDataset(data=s1)
>>>
>>> # mandatory properties are also in metadata
... x.creator = "Me, myself and I"
>>> print(x.creator)  # == "Me, myself and I"
Me, myself and I
>>> # This is also changed
... print(x.meta["creator"])  # == "Me, myself and I"
Me, myself and I
```

Product x.toString() :
```
# Product
# description = "product example with several datasets"
# meta = MetaData{[description = product example with several datasets,
creator = Me, myself and I, creationDate = 2000-01-01T00:00:00.000000 TAI(0),
instrument = Crystal-Ball, startDate = , endDate = , rootCause = UNKNOWN,
modelName = Mk II, type = UNKNOWN, mission = SVOM, ], listeners = [Product
7696577608224 "product example with several datasets", ]}
# History
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# data =

# data =

# [ RawImage ]
# ArrayDataset
# description = "image1"
# meta = MetaData{[], listeners = []}
# unit = "ev"
# data =

1 4 7
2 5 8
3 6 9

# [ QualityImage ]
# ArrayDataset
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# unit = "None"
# data =

0.1 4000.0 -2
0.5 60000000.0 0
0.7 8 3.1

# [ Spectrum ]
# TableDataset
# description = "UNKNOWN"
# meta = MetaData{[], listeners = []}
# data =

# col1 col2
# eV cnt
1 0
4.4 43.2
5400.0 2000.0
```



ArrayDataset

creation:
```
>>> a1 = [1, 4.4, 5.4E3]      # a 1D array of data
>>> a2 = 'ev'                 # unit
>>> a3 = 'three energy vals'  # description
>>> v = ArrayDataset(data=a1, unit=a2, description=a3)
>>> # simpler but error-prone
>>> v1 = ArrayDataset(a1, a2, description=a3)
>>> print(v)
ArrayDataset{ description = "three energy vals", meta =
MetaData[], data = "[1, 4.4, 5400.0]", unit = "ev"}
>>>
>>> print(v == v1)
True
```

data access:
```
>>> v1.data = [34]
>>> v1.unit = 'm'
>>> print('The diameter is %f %s.' % (v1.data[0], v1.unit))
The diameter is 34.000000 m.

>>> # iteration
>>> i = []
>>> for m in v:
...     i.append(m)
...
>>> #assert i == a1
>>> print(i)
[1, 4.4, 5400.0]
```

toString():
```
# make a 4-D array: a list of 2 lists of 3 lists of 4 lists
of 5 elements.
s = [[[[i + j + k + l for i in range(5)] for j in range(4)]
for k in range(3)] for l in range(2)]
x = ArrayDataset(data=s)
print(x.toString())
```

output

ArrayDataset x.toString():
```
# ArrayDataset
# description = "UNKNOWN"
# meta = MetaData{[],
listeners = []}
# unit = "None"
# data =

0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7


1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8


2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
6 7 8 9


#=== dimension 4

1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8


2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
6 7 8 9


3 4 5 6
4 5 6 7
5 6 7 8
6 7 8 9
7 8 9 10


#=== dimension 4
```

Product Access Layer (pal)

Product Access Layer provides classes for the storing, retrieving, tagging, and context creating of data product modeled in the dataset package. This package lets one store data in logical ``pools'', and makes the data accessible with light weight product references. A ProductStorage interface is provided to handle saving/retrieving/querying data in registered ProductPools.

In a data processing pipeline or network of processing nodes, data products are generated within a context. Data processers, data storages, and data consumers often need to have relevant context data recorded with a product. However the context could have a large size so including them as metadata of the product is often impractical.

Once a data product is saved by ProductStorage it can have a reference generated for it. Through its reference the product can be accessed. The size of such references are typically less than a hundred bytes, like a URL. References enable SPDCs to encapsulate rich, deep, sophisticated, and accessible contextual data, yet remain light weight.

This package implements a data structure modeled after *Herschel Common Software System (v15)* products (https://www.cosmos.esa.int/web/herschel/data-products-overview/}, taking other requirements of scientific observation and data processing into account. The APIs are kept as compatible with HCSS/HIPE (Riedinger 2009, Ott 2010) as possible, with descriptions treated as part of interface contract.

pal

Create a product and a productStorage with a pool registered
```
>>> # a pool for demonstration will be
create here
... demopoolpath = '/tmp/demopool'
>>> demopool = 'file://' + demopoolpath
>>> # clean possible data left from
previous runs
... os.system('rm -rf ' + demopoolpath)
>>> # create a prooduct
... x = Product(description='in store')
>>> print(x)
{meta = "MetaData['description',
'creator', 'creationDate',
'instrument', 'startDate', 'endDate',
'rootCause', 'modelName', 'type',
'mission']", _sets = [], history =
{meta = "MetaData[]", _sets = []}}
>>> # create a product store
... pstore =
ProductStorage(pool=demopool)
```

Save the product and get a reference
```
>>> prodref = pstore.save(x)
>>> # create an empty mapcontext
... mc = MapContext()
>>> # put the ref in the context.
... mc['refs']['very-useful'] = prodref
>>> # get the urn string
... urn = prodref.urn
>>> print(urn)
urn:file:///tmp/demopool:Product:0
```

re-create a product only using the urn
```
>>> newp = getProductObject(urn)
>>> # the new and the old one are equal
... print(newp == x)
True
```

Processing Node Server (pns)

This Web API Server for a data processing pipeline/network node provides interfaces to configure the data processing task software (PTS) in a processing node, to make a run request, to deliver necessary input data, and to read results, all via RESTful web APIs.

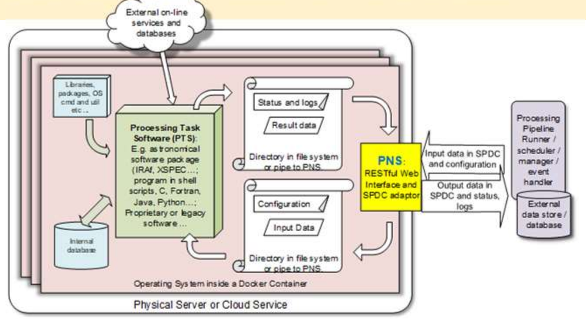
Many data processing pipelines need to run software that only runs on a specific combination of OS type, version, language, and library. These software could be impractical to replace or modify but need to be run side-by-side with software of incompatible environments/formats to form an integral data processing pipeline, each software being a ``node'' to perform a processing task. Docker containers are often the perfect solution to run software with incompatible dependencies.

PNS installed on a Docker container or a normal server allows a processing tasks to run in the PNS memory space, in a daemon process, or as an OS process receiving input and delivering output through a ``delivery man'' protocol.

SPDC v0.8 test suite has been run on CentOS, Ubuntu, and Cygwin with Apache and Flask servers. The client-server pipeline architecture is shown to work with a server running astronomical data processing software written in C, FORTRAN, Python using Pyraf and Anaconda.

Astronomers develop the Processing Task Software (**PTS**) in his/her favorite environment, using tools and libraries. All are possibly incompatible with those of other modes.

PTS is stateless -- the result can be calculated and reproduced with given input and configuration. PTS does not remember previous runs.


External on-line services and databases
Libraries, packages, OS cmd and util etc...
Processing Task Software (PTS): E.g. astronomical software package (IRAf, XSPEC...; program in shell scripts, C, Fortran, Java, Python...; Proprietary or legacy software ...
Internal database
Status and logs
Result data
Directory in file system or pipe to PNS
Configuration
Input Data
Directory in file system or pipe to PNS
PNS: RESTful Web Interface and SPDC adaptor
Input data in SPDC and configuration
Output data in SPDC and status, logs
Processing Pipeline Runner / scheduler / manager / event handler
External data store / database
Operating System inside a Docker Container
Physical Server or Cloud Service

The **Processing Node Server (PNS)** passes input data and configuring information from the pipeline runner to the PTS, runs the PTS (successfully or not) , collects the results, and returns them to the pipeline runner.

References
Ott, S. 2010, in Astronomical Data Analysis Software and Systems XIX, edited by Y. Mizumoto, K. I. Morita, & M. Ohishi, vol. 434 of Astronomical Society of the Pacific Conference Series, 139. 1011.1209
Riedinger, J. 2009, ESA Bulletin, 139, 14