

psana - Migration from pyana



Unknown macro: 'html'

Recommended Next Topics

- [psana](#)



Unknown macro: 'html'



Unknown macro: 'html'

- [Introduction](#)
- [Data types and access methods](#)
- [Difference in configuration file](#)
- [Constructor arguments and configuration data](#)
 - [Pyana](#)
 - [Psana](#)
- [Access to data in pyana and psana](#)
 - [Event data access in pyana](#)
 - [Event data access in psana](#)
 - [Access to configuration objects in pyana](#)
 - [Access to configuration objects in psana](#)
- [Differences in data classes](#)
 - [Acqiris](#)
 - [Bld](#)
 - [CsPad](#)
 - [CsPad2x2](#)
 - [EPICS](#)
 - [Evr](#)
 - [Pnccd](#)
 - [Princeton](#)
 - [Imp](#)
 - [Ipimb](#)
 - [Timepix](#)
 - [Opal1k](#)
 - [All other modules](#)
- [Interactive psana](#)
- [References](#)



Unknown macro: 'html'

Introduction

Offline analysis software for LCLS data historically pass a few stages;

- [myana](#) – the first C-based package,
 - [pyana](#) – python-based framework,
 - [psana - Original Documentation](#) – C++-based framework.
- Currently, features of both frameworks are combined together under the unified
- [psana - Original Documentation](#) – framework, which works both with C++ and python modules.
- This new framework allows data processing using advantage of both C++ and python modules. However, there is no full backward compatibility between new [psana - Original Documentation](#) framework and [pyana](#) modules. Original [pyana](#) modules would not work directly in [psana - Original Documentation](#), they need to be changed. In this note we discuss the difference between [pyana](#) and [python-psana - Original Documentation](#) modules and present the references to documentation, which help to migrate from [pyana](#) to [python-psana - Original Documentation](#).

Data types and access methods

LCLS data types have different implementation in pyana and python-psana. Auto-generated from source code documentation is available in

- <https://pswww.slac.stanford.edu/swdoc/releases/ana-current/pyana-ref/html/> for pyana and
- <https://pswww.slac.stanford.edu/swdoc/releases/ana-current/psana-ref/html/> for python-psana.

Code of examples can be found in the packages

- https://pswww.slac.stanford.edu/trac/psdm/browser/psdm/pyana_examples#trunk/src - pyana_examples, and
 - https://pswww.slac.stanford.edu/trac/psdm/browser/psdm/psana_examples#trunk/src - psana_examples.
- From this documentation it is clear that there is no full backward compatibility between these libraries; data types and access methods are different. Below we present a list of references with code examples for both frameworks.

Difference in configuration file

Configuration file in both frameworks is starting from section `pyana` or `psana modules` followed by the list of parameters. The list of parameters is different for historical reasons. When python modules are designed to support both frameworks, both sections may be presented in the same configuration file. Alien section is ignored at run time of each framework. Example of these sections in the same configuration file:

```
[pyana]
files = /reg/d/psdm/xcs/xcs72913/xtc/e265-r0049-s00-c00.xtc /reg/d/psdm/xcs/xcs72913/xtc/e265-r0049-s04-c00.xtc /reg/d/psdm/xcs/xcs72913/xtc/e265-r0049-s05-c00.xtc
num-events = 500
#skip-events = 0
#num-cpu = 1
verbose = 1
modules = py_img_algos.tahometer py_img_algos.cspad_arr_producer py_img_algos.cspad_image_producer py_img_algos.image_save_in_file

[psana]
files = exp=xcs72913:run=49
events = 500
#skip-events = 0
modules = py_img_algos.tahometer py_img_algos.cspad_arr_producer py_img_algos.cspad_image_producer py_img_algos.image_save_in_file
```

A few differences can be clearly seen:

- **psana** accepts both forms for `files`, **pyana** - only explicit list of files.
- Parameter for number of events has different name `events` and `num-events` for **psana** and **pyana**, respectively.
- `verbose` mode currently works for **pyana** only. Logger does not have interface in **psana** yet.
- **psana** currently does not support multiprocessor mode `num-cpu`.

Constructor arguments and configuration data

Pyana

Original pyana design used constructor arguments to pass configuration parameters to user modules. Typical user module defined its `__init__` method with a number of arguments with many or all of them having default values:

```
class user_module(object):

    def __init__(self, source, threshold="0.25", max_count="1000"):
        self.source = source
        self.threshold = float(threshold)
        self.max_count = int(max_count)
```

The arguments that do not provide default values must be specified in the configuration file, arguments with the default values can be overridden by the values from configuration file. Suppose that configuration files contains this:

```
[user_package.user_module]
source = SrcString
threshold = 1.25
```

Then **psana** will instantiate user module with the parameters `user_module(source="SrcString", threshold="1.25")` and `max_count` argument will use default value from method declaration. Note that all arguments are passed as string values, no conversion to integers or floating point numbers is performed.

Starting with the release `ana-0.9.6` **pyana** was modified to support different style of access to configuration information. In addition to receiving configuration parameters through constructor arguments one can also use few special methods of the module class to obtain the values of the parameters. These methods are:

<code>self.configBool(param_name[, default])</code>	returns value of parameter as a boolean value, strings "yes", "true", "True", "on", "1" represent true value, strings "no", "false", "False", "off", "0" represent false value, any other string will raise exception. If parameter is not defined in a file then default value is returned without conversion, if default value was not given then exception is raised.
---	--

<code>self.configInt(param_name[, default])</code>	returns value of parameter as integer value. If parameter is not defined in a file then default value is returned without conversion, if default value was not given then exception is raised. If conversion from string to integer fails the standard exception is raised.
<code>self.configFloat(param_name[, default])</code>	returns value of parameter as floating point value. If parameter is not defined in a file then default value is returned without conversion, if default value was not given then exception is raised. If conversion from string to floating point fails the standard exception is raised.
<code>self.configStr(param_name[, default])</code>	returns value of parameter as string. If parameter is not defined in a file then default value is returned without conversion, if default value was not given then exception is raised.
<code>self.configSrc(param_name[, default])</code>	returns value of parameter as "source address" string. If parameter is not defined in a file then default value is returned without conversion, if default value was not given then exception is raised. This method does the same as <code>self.configStr</code> but it may also check correctness of the source string format.
<code>self.configListBool(param_name)</code>	returns value of parameter as a list of boolean values. If parameter is not defined in a file then empty list is returned, otherwise every word in a parameter value is converted to boolean (according to same rules as defined for <code>self.configBool</code>) and all values are returned in one list.
<code>self.configListInt(param_name)</code>	returns value of parameter as a list of integers. If parameter is not defined in a file then empty list is returned, otherwise every word in a parameter value is converted to integers and all numbers are returned in one list. Conversion errors will raise exception.
<code>self.configListFloat(param_name)</code>	returns value of parameter as a list of floating point numbers. If parameter is not defined in a file then empty list is returned, otherwise every word in a parameter value is converted to float and all numbers are returned in one list. Conversion errors will raise exception.
<code>self.configListStr(param_name)</code>	returns value of parameter as a list of strings. If parameter is not defined in a file then empty list is returned, otherwise parameter value is split into words which are returned in one list.
<code>self.configListSrc(param_name)</code>	returns value of parameter as a list of "source address" strings. If parameter is not defined in a file then empty list is returned, otherwise parameter value is split into words which are returned in one list.

These methods are "magic" in a sense that they are not defined by the module itself but instead are generated by the framework.

As the values of the configuration parameters can be obtained from the above methods defining arguments in the constructor is no longer necessary. pyana now supports modules which have empty argument list of the `__init__` method (except for `self` argument). The constructor of the example module above can be now defined as:

```
class user_module(object):

    def __init__(self):
        self.source = self.configStr('source')      # no default - must be defined in a config file
        self.threshold = self.configFloat('threshold', 0.25)
        self.max_count = self.configInt('max_count', 1000)
```

This style should be preferred in pyana and it provides easier migration to psana.

Psana

The situation with the arguments and configuration parameters in psana is opposite to the pyana situation - Python modules in psana are not supposed to define any arguments for the constructor and they should use methods defined above to access values of the configuration parameters. Like in pyana these methods are "magic" as they are defined by the framework itself and not by the module itself or its base classes. Unlike in pyana the methods `self.configSrc()` and `self.configListSrc()` return instances of [psana.Source class](#) instead of string, this class is used as an argument to `evt.get()` method in psana.

As a compatibility feature psana supports "old pyana style" of the constructor which receives configuration parameters via constructor arguments. This means that old pyana modules can work unchanged in this respect (but will still require changes for data access). This compatibility feature may be removed in the future as it may have unexpected implications, so it is much better to switch to the new psana style, especially that pyana now also supports this style.

Access to data in pyana and psana

There are very significant differences between pyana and psana in how the data are retrieved from the framework (via event and environment structures) and the format of the data classes. psana shares framework and data definition with C++ implementation and many things are closer to C++ interface than to the pyana (which is inspired by pdsdata). Here we discuss differences in retrieving the data.

Event data access in pyana

Pyana was designed based on pdsdata and myana and originally provided methods of data access which looked similar to those defined in myana. In particular event object provided a [number of specialized methods](#) for some (but not all) individual data types, e.g. `evt.getAcqValue(address, channel, env)` for accessing Acqiris data, `evt.getFeeGasDet()` for accessing FEE gas detector data. In addition to these specialized methods there is a generic method `evt.get(typeId, address:string)` which can be used to retrieve any type of data. There are two "overloads" of this method:

1. `evt.get(typeId:int, address:string)` which takes integer number for the first argument and "source address" string for second argument. Integer number is an XTC [TypeId](#) value such as `xtc.TypeId.Type.Id_Frame`. This method retrieves detector data of the type corresponding to the [TypeId](#) coming from a device that matches source address.
2. `evt.get(key:string[, default=None])` which takes a string for the first argument and optionally any object for second argument. This method retrieves "user data" object from event which was stored in the event by user module with by calling `evt.put(object, key)` with the same string key.

Best way to access event data in pyana is to use generic `evt.get()` method and not specialized methods, it would be much easier to migrate this code later to psana. Here is an example of this:

```
from pypdsdata import xtc
.....
    src = self.configSrc('source', '')
    data = evt.get(xtc.TypeId.Type.Id_FEEGasDetEnergy, src)
    if data:
        print " f_11_ENRC =", data.f_11_ENRC
```



The list of currently available [TypeId](#) can be seen in the file [TypeId.cpp](#).
The list of [Source](#)-s available in xtc file can be obtained by the command:

```
psana -m EventKeys <path-to-xtc-file>
```

Event data access in psana

Psana does not provide any specialized methods for retrieving individual data types, instead it provides generic `evt.get()` method whose [interface](#) is much closer to that of C++ class, though it also provides few overloads for pyana compatibility. There are several "overloads" of this method depending on the number and types of parameters:

1. `evt.get(type, src[, key:string])` where `type` is python class corresponding to the data type (or list of classes), e.g. `psana.Camera.FrameV1`. The `src` argument must be an instance of `psana.Source` type which provides a match for device address, `psana.Source` is returned from `self.configSrc(...)` method. If `key` argument is provided then it must be a string, missing `key` argument is identical to empty string.
2. `evt.get(type[, key:string])` - variation of the above method without source argument. This method is used to retrieve data which is not associated with any device but is a property of the event as a whole. Good example of this kind of data is the `psana.EventId` object.
3. `self.get(typeId:int, addr:string)` - this is pyana-compatibility method and should only be used during migration or in the code which is supposed to run in both pyana and psana. This method is equivalent to pyana method `evt.get(TypeId, address)`. Note that in psana there will be data types which do not have corresponding [TypeId](#), this method cannot be used with those types ([TypeId](#) is a property of XTC data types, other data types do not have [TypeId](#)).
4. `self.get(key:string)` - this is pyana-compatibility method and should only be used during migration or in the code which is supposed to run in both pyana and psana. This method is equivalent to pyana method `evt.get(key, None)` and is used to retrieve data stored with `evt.put(object, key)`.



The list of currently available [TypeId](#) can be seen in the file [TypeId.cpp](#).
The list of `type` and `src` arguments available in xtc file can be obtained by the command:

```
psana -m EventKeys <path-to-xtc-file>
```

The type name printed like `Psana::Ipimb::DataV2` should be used in code as `psana.Ipimb.DataV2...`

psana example corresponding to the above pyana example which retrieves gas detector data should look like:

```
# this is psana code, will not work in pyana
import psana
....
    src = self.configSrc('source', '')
    data = evt.get(psana.Bld.BldDataFEEGasDetEnergy, src)
    if data:
        print "  f_11_ENRC =", data.f_11_ENRC()    # note method call
```



Specific data attributes, like `f_11_ENRC()`, can be found in the code reference for [psana](#) and [pyana](#), respectively.

If one wants to write the code that should work in both pyana and psana then some modification of the pyana example should do:

```
from pydpdsdata import xtc
.....
    src = self.configSrc('source', '')
    data = evt.get(xtc.TypeId.Type.Id_FEEGasDetEnergy, src)
    if data:
        if env.fwkName() == 'pyana':
            print "  f_11_ENRC =", data.f_11_ENRC
        else:
            print "  f_11_ENRC =", data.f_11_ENRC()
```

Note that there is a code specific to each framework and this is related to the differences in the data object interfaces.

Access to configuration objects in pyana

Just like event class the [environment class](#) in pyana provides a number of specific method for individual data types and a generic method which works for any data type. Specific methods should not be used frequently, one should prefer to use generic method. Generic method has this definition:

- `env.getConfig(typeId:int[, address=None])` - takes integer number for the first argument and optional string for second. Integer number is an XTC Typeld value such as `xtc.TypeId.Type.Id_AcqConfig` (note that [Typeld](#) values for event data and configuration data are different). This method retrieves configuration data of the type corresponding to the [Typeld](#) coming from a device that matches source address.

An example:

```
from pydpdsdata import xtc
.....
    src = self.configSrc('source', '')
    config = env.getConfig(xtc.TypeId.Type.Id_AcqConfig, src)
    if config:
        print "config.nbrChannels =", config.nbrChannels()
```

Access to configuration objects in psana

To access configuration data in psana there are two methods available: "true psana" and pyana-compatibility.

For code which runs in psana only it is better to use "true psana" method which consists in first getting access to special "config store" object and second retrieving data from that store object. To access store object one should call `env.configStore()` method without parameters which returns the instance of the [EnvObjectStore](#) class. This class defines `get()` method which is a simplified version of `evt.get()` method and has these overloads:

1. `store.get(type, src)` where `type` is python class corresponding to the configuration data type (or list of classes), e.g. `psana.Acqiris.Config`. `src` argument must be an instance of `psana.Source` type which provides a match for device address.
2. `self.get(typeId:int[, address:string=""])` - this is pyana-compatibility method and should only be used in the code which is supposed to run in both pyana and psana. This method is equivalent to pyana method `env.getConfig(TypeId, address)`.

For pyana compatibility environment also defines `evt.getConfig(...)` method which is a shortcut for `env.configStore().get(...)`, this method should only be used in a code which should run in both pyana and psana.

Here is a psana example for accessing the same Acqiris configuration data:

```
import psana
.....
    src = self.configSrc('source', '')
    config = env.configStore().get(psana.Acqiris.Config, src)
    if config:
        print "config.nbrChannels =", config.nbrChannels()
```

Previous pyana example is supposed to work in psana without change.

Differences in data classes

Tables below provide summary of differences between data objects inside pyana and psana.

For pyana the pieces of code imply that following import statement appears somewhere in the module file:

```
from pypdsdata import xtc
```

for psana this import should be replaced with

```
from psana import *
```

In pyana `evt.get()` and `env.getConfig()` accept string for `source` argument while in psana the type of the argument must be `psana.Source`. In both pyana and psana the source address can be set with `self.configSrc()` as this method returns correct type in each framework:

```
self.m_src = self.configSrc('source', '')
```

As it is discussed above pyana code can be run inside psana in compatibility mode. In this case the code needs changes only if data classes provide different interfaces.

Acqiris

pyana	psana
_pdsdata.acqiris	Psana.Acqiris
<code>config = env.getConfig(xtc.TypeId.Type.Id_AcqConfig, self.m_src)</code>	<code>config = env.configStore().get(Acqiris.Config, self.m_src)</code>
<code>acqData = evt.get(xtc.TypeId.Type.Id_AcqWaveform, self.m_src)</code>	<code>acqData = evt.get(Acqiris.DataDesc, self.m_src)</code>
access to data also differs...	

Bld

pyana	psana
_pdsdata.bld	Psana.Bld
<code>data = evt.get(xtc.TypeId.Type.Id_FEEGasDetEnergy, self.m_src)</code>	<code>data = evt.get(Bld.BldDataFEEGasDetEnergy, self.m_src)</code>
<code>data = evt.get(xtc.TypeId.Type.Id_EBeam, self.m_src)</code>	<code>data = evt.get(Bld.BldDataEBeam, self.m_src)</code>
<code>data = evt.get(xtc.TypeId.Type.Id_PhaseCavity, self.m_src)</code>	<code>data = evt.get(Bld.BldDataPhaseCavity, self.m_src)</code>
<code>data = evt.get(xtc.TypeId.Type.Id_GMD, self.m_src)</code>	<code>data = evt.get(Bld.BldDataGMD, self.m_src)</code>

CsPad

pyana	psana
-------	-------

_pdsdata.cspad	Psana.CsPad
<code>config = env.getConfig(TypeId.Type.Id_CspadConfig, self.m_src)</code>	<code>config = env.configStore().get(CsPad.Config, self.m_src)</code>
<code>quads = evt.getTypeId.Type.Id_CspadElement, self.m_src)</code>	<code>data = evt.get(CsPad.Data, self.m_src)</code>
<code>for q in quads : data = q.data() # image data as 3-dimentional array of the shape (N, 185, 388)</code>	
<code>list_of_masks = [config.roiMask(q) for q in range(4)]</code>	<code>list_of_masks = [config.roiMask(q) for q in range(4)]</code>
<code>print " numQuads =", config.numQuads();</code>	<code>nQuads = data.quads_shape()[0] # etc.</code>
<code>for i in range(nQuads): q = data.quads(i); data_arr = q.data() # etc.</code>	<code>for i in range(nQuads): q = data.quads(i); data_arr = q.data() # etc.</code>

CsPad2x2

pyana	psana
_pdsdata.cspad2x2	Psana.CsPad2x2
<code>config = env.getConfig(xtc.TypeId.Type.Id_Cspad2x2Config, self.m_src)</code>	<code>(works the same) config = env.getConfig(CsPad2x2Config, self.m_src)</code>
<code>print " roiMask =", config.roiMask()</code>	<code>roiMask = config.roiMask()</code>
<code>elem = evt.get(xtc.TypeId.Type.Id_Cspad2x2Element, self.m_src)</code>	<code>elem = evt.get(CsPad2x2.Element, self.m_src)</code>
<code>data = elem.data() # image data as 3-dimentional array of the shape (185, 388, 2)</code>	<code>data = elem.data()</code>
<code>{{}}</code>	<code>{{}}</code>

EPICS

pyana	psana
_pdsdata.epics	Psana.Epics
<code>{{}}</code>	<code>{{}}</code>

Evr

pyana	psana
_pdsdata.evr	Psana.EvrData
<code>{{}}</code>	<code>{{}}</code>

Pncdd

pyana	psana
_pdsdata.pncdd	Psana.PNCCD
<code>{{}}</code>	<code>{{}}</code>

Princeton

pyana	psana
_pdsdata.princeton	Psana.PNCCD
<code>{{}}</code>	<code>{{}}</code>

Imp

pyana	psana
-------	-------

_pdsdata.imp	Psana.Imp
<code>self.m_src = self.configSrc('source', ':Imp')</code>	<code>self.m_src = self.configSrc('source', ':Imp')</code>
<code>config = env.getConfig(xtc.TypeId.Type.Id_ImpConfig, self.m_src)</code>	<code>config = env.configStore().get(Imp.Config, self.m_src)</code>
<code>data = evt.get(xtc.TypeId.Type.Id_ImpData, self.m_src)</code>	<code>data = evt.get(Imp.Element, self.m_src)</code>
<code>print "TrigDelay =", config.get(imp.ConfigV1.Registers.TrigDelay)</code>	<code>print " trigDelay =", config.trigDelay() # etc.</code>
<code>print "frameNumber =", data.frameNumber()</code>	<code>print " frameNumber =", data.frameNumber() # etc.</code>

Ipimb

pyana	psana
_pdsdata.ipimb	Psana.Ipimb
{}	{}

Timepix

pyana	psana
_pdsdata.timepix	Psana.Timepix
{}	{}

Opal1k

pyana	psana
_pdsdata.opal1k	Psana.Opal1k
{}	{}

All other modules

pyana	psana
_pdsdata modules	Psana modules

Interactive psana

Interactive psana is available beginning from [release](#) ana-0.9.0.
Example of how to start interactive python-psana:

```
% ipython
from psana import *
help(psana.CsPad)
```

References

- Auto-generated documentation for pyana modules: <https://pswww.slac.stanford.edu/swdoc/releases/ana-current/pyana-ref/html/>
- Auto-generated documentation for python modules in psana: <https://pswww.slac.stanford.edu/swdoc/releases/ana-current/psana-ref/html/>
- Package pyana_examples: https://pswww.slac.stanford.edu/trac/psdm/browser/psdm/pyana_examples#trunk/src
- Package psana_examples: https://pswww.slac.stanford.edu/trac/psdm/browser/psdm/psana_examples#trunk/src

ples: https://pswww.slac.stanford.edu/trac/psdm/browser/psdm/psana_examples#trunk/src