i3py Documentation

Release 0.1.0-dev

I3py Authors

Aug 28, 2019

Contents

I3py is a Python package for instrument interfacing.

CHAPTER 1

User guide for I3py

CHAPTER 2

Extending I3py

2.1 Driver machinery

I3py strives to provide a convenient but flexible interface to write drivers that can handle the full complexity of the instrument you need to interface. Of course this has some cost in term of internal complexity, but those are believed to be worth it.

This section of the documentation will first take you through the notions used in I3py to structure your driver, and how to use them to write your driver. It will try to remain generic but will use the case of instruments controlled through the VISA protocol to illustrate the examples as it is a very common case.

Once this done, the following sections will dive in the detailed working of the different components making up a driver, and will explore in more details some of the notions previously presented. The reading of those sections is not mandatory to write a driver but will help you understand the inner working of I3py and help you tackle more challenging driver designs.

2.1.1 Writing a driver

This section will focus on giving you a quick overview of the tools available to you to write your driver. While it should be sufficient to write simple drivers you may want to check the following sections for more detailed explanations.

Note: It may seem obvious, but, in order to write a driver for your instrument you need both a good knowledge of the instrument and to read its manual. Finally you need to have access to the real instrument as the manual is likely to be elliptic if not incorrect in some cases (and it is perfectly possible for your instrument firmware to be buggy). Finally in the early stages of development you should make sure you can communicate with your instrument using a third party tool such as NiMAX for VISA instruments or the vendor provided software in other cases.

Contents

General organization

I3py drivers are organized along the following concepts:

- the “driver” is at the highest level, it is responsible for handling the communication with the physical instrument.
- the parameters of the instrument that can be read and possibly set by the user, and whose value does not change spontaneously (any measured quantity typically falls outside this category) are described using “features”, which are advanced and highly customizable descriptors (ie properties, please refer to Python documentation if you do not know what a property is).
- the operations that an instrument can perform (such as a measure), are described using an “action” that is a simple wrapper around a method. It provides optionally some validation/conversion of the argument and return values.
- finally drivers can be structured in “subsystems” and multiple channels or cards inside a rack can be described using a “channel”, which is nothing else than a subsystem with an attached id.

All the above concepts will be illustrated below on concrete examples.

Creating the driver

In I3py, each driver is defined in a class declaration. All drivers should inherit from the `BaseDriver` class, however most of the time you will most probably use a derived class handling the low-level communication with the instrument such as `i3py.backends.visa.message_based.VisaMessageDriver`.

As I3py strives for uniformity in the drivers, it provides for some instruments a standardized base class defining the expected features and actions. As those ‘standards’ cannot assume any communication mean with the instrument, to use those your driver should inherit from a base class handling the communication and all the ‘standards’ your instrument supports.

```
class MyDriver(VisaMessageDriver, IEEEIdentify):
    """My driver (supporting *IDN?) docstring.

    """
    pass
```

If no base class provides the proper communication method to read and write the values of the instrument features, you will have to implement the following methods:

- `default_get_feature()`: This method is charged with querying the value of the feature. See the API docs for more details.
- `default_set_feature()`: This method is charged with setting the value of the feature. See the API docs for more details.
- `default_check_operation()`: This method is charged with checking that setting a feature did go properly. This method is free to use any method it see fit to perform this check. See the API docs for more details.

For more details about what are standards and how to use them please refer to *Standards*

Note: To simplify the handling of changes to driver over time I3py enforce to add a class variable __*version*__ to all drivers. The format of this variable should be the one of a usual version string: MAJOR.MINOR.MICRO

Adding a Feature

A Feature describes a property of the instrument, that, as already mentioned does not change in a spontaneous way. This restriction comes from the fact that the values of features are cached. They can and should be discarded when some other setting of the driver is modified, but not in a spontaneous manner.

To add a Feature to your instrument you have nothing else to do that assign a Feature subclass to an identifier. As illustrated in the example below :

```
class MyDriver(VisaMessageDriver, IEEEIdentify):
    """My driver (supporting *IDN?) docstring.

    """
    mode = Unicode('MODE?', 'MODE {}', values=('CW', 'PULSED'))
```

The first argument is the command to get the value of the feature, the second the command to set it. For message based VISA driver, this is the true SCPI command string and the braces will be filled with the set value.

Additional keywords are used to customized the action taken before getting/setting a value (such as checking this is allowed), how to extract the value or how to validate that the value is meaningful for the instrument.

When subclassing an existing instrument it is often possible that a feature already exists on the parent class but is not properly configured (for example the values are incorrect). In such cases, it is not necessary to entirely redefine the feature, one can use `set_feat` to change the proper keyword arguments values.

```
class MyNewDriver(MyDriver):
    """My driver (supporting *IDN?) docstring.

    """
    mode = set_feat(values=('CW', 'PULSED', 'TRIGGERED'))
```

The detailed working of Features is detailed in *Features*, and all the existing features are described in the API. Finally, as instruments can be often quite surprising in their behaviors, the default behaviors of the provided features may prove insufficient. More complex customization are possible and detailed in the section *Advanced customization* of this manual.

Adding an Action

Actions are light wrapper around methods. They provide similar facility to run checks and conversion on the input and output values as do features. To declare one, you only have to declare a method:

```
class MyDriver(VisaMessageDriver):
    """My driver (supporting *IDN?) docstring.

    """

    @Action(values={'kind': ('volt', 'curr')})
    def read_state(self, kind):
        """Read the instrument state.

        """
        pass
```

The above example shows how to check the value of an argument is valid.

The detailed working of actions is described in *Actions* section. Just like features several classes of actions exist and are describe in the API. Actions support advanced customization just like features which are described in section *Advanced customization*

Note: To improve readability and allow third-party tool to improve the integration of I3py drivers, it recommended to provide type annotations for Action and in particular for their return values.

Using subsystem and channels

Subsystems allow to group features into coherent ensemble, which can allow to avoid ridiculously long names. For example many lock-in amplifiers include a built-in oscillator and subsytems allow for example to group the related features such as amplitude and frequency as shown below:

```
class MyDriver(VisaMessageDriver):
    """My driver (supporting *IDN?) docstring.

    """

    oscillator = subsystem()
    with oscillator as o:

        o.frequency = Float('OSC:FREQ?', 'OSC:FREQ {}')
```

Actions can also be attached to a subsystems:

```
class MyDriver(VisaMessageDriver):
    """My driver (supporting *IDN?) docstring.

    """

    oscillator = subsystem()
    with oscillator as o:

        @o
        @Action()
```

(continues on next page)

(continued from previous page)

```
        def is_sync(self):
            pass
```

By default, a subsystem is a subclass of :py:class: *~i3py.core.base_subsystem.SubSystem* and any subsystem of the parent class of the driver. You can specify additional base classes as a tuple passed as first argument to subsystem.

Channels are similar to subsytems but are identified by an id as an instrument may have an arbitrary number of channel of the same kind (the default class is `Channel`). Adding a channel to an instrument is similar to adding a subsystem save that one must specify what are the valid channel ids. One can specify a static list of ids or a the name of a method on the parent listing the available channels when called. This method should take no argument. Furthermore one can declare, aliases for the channel ids to provide more user friendly name than the ones used by the driver.

For more details please refer to the API documentation or to the dedicated section of the documentation about subsystems and channels *Subsystems and channels*.

Handling options

Depending on the instrument firmware or the option bought, some capabilities of the instrument may not be available. To reflect this reality, i3py allows to define special features `Options` used to recover the instrument options.

Note: A single driver can define multiple options features but they all use different names.

Features, actions, subsystem and channel all support a dedicated keyword argument 'options' to specify tests to perform on the instrument option before granting the user access to it for the first time. The full check is only performed the first time since options are meant to describe hardware or firmware settings that cannot change during the instrument operation. The format in which to specify the checks is the following:

> 'feature_options_name['option_name'] == option_value'

Actually any valid boolean assertion can be evaluated so if an option can only be True or False the equality test is useless. Furthermore multiple tests can be separated by ";" .

Note: Operation that cannot be performed at runtime because the instrument is not properly configured fall outside the scope of options and should be inhibited if necessary using the 'checks' mechanism that exists on features, actions, subsystems and channels.

For more details please refer to the API documentation.

Special class variables for VISA based driver

In the case of VISA based drivers, it is desirable to specify which communication interfaces are supported by the instrument, along which the parameters to use (such as termination characters, which may differ between interfaces). All those information can be specified to I3py drivers through the use of the class level variables listed below.

Note: To avoid polluting the driver namespace, VISA specific method are grouped in the visa_resource subsystem, accessible from the top level driver.

INTERFACES

Dictionary specifying the interfaces supported by the instrument. For each type of interface a dictionary (or a list of dictionary), specifying the default arguments to use should be provided. Valid interfaces are :

- ASRL: serial interface (RS232)
- USB: usb interface
- TCPIP: ethernet based interface
- GPIB: gpib based interface
- PXI: pxi based interface
- VXI: vxi based interface

For each supported interface, the dictionary should contain at least the resource class to use. In addition, it can contain interface specific settings that users will not have to provide to start the driver. For example, if the instrument support the using raw sockets on TCPIP, the port number is required and can be specified as follow.

```
INTERFACES = {'TCPIP': {'resource_class': 'SOCKET',
                        'port': '50000'}}
```

The valid keys for each interface matches the named used in VISA resource names which are described in PyVISA documentation.

DEFAULTS

Dictionary specifying the default parameters to use for the VISA session. As some of those can be interface or resource specific, the valid keys for the dictionary include any pair (interface_type, resource_class), any interface_type, any resource_class and *'COMMON'* that applies to all interfaces/resources. The values associated to each key is expected to be a dictionary, whose keys match the attributes of the underlying VISA resource. The most commons are:

- write_termination: character appended at the end of each sent message
- read_termination: character expected at the end of each received message.
- timeout: time in ms after which to consider that the communication failed.

NON_VISA_NAMES

By default all arguments passed to a VISA driver are used to build the resource name. This class holds a tuple of named reserved to other use. By default it is set to *('parameters', 'backend')*, which should be sufficient be sufficient in most cases.

'parameters' is a dictionary whose content is by default passed to the underlying PyVISA object, but it is a matter of simply overriding initialize to handle it in a different fashion.

Note: Serial instruments usually requires to be switched to remote control before accepting any instruction. To streamline this process, the `i3py.drivers.common.rs232.VisaRS232` provides the automatic addition of the proper header to all outgoing messages when one connect to the instrument through its serial interface (and only then). The header to use can be specified as a BYTE string using the *RS232_HEADER* class variable.

Making the driver accessible from the top level manufacturer package

Drivers in I3py are organized by manufacturers (inside each manufacturer package, they can be organized by instrument type). However because building the driver class is more expensive than regular Python classes, I3py provides a way to make drivers visible from the top level manufacturer package that does not lead to their automatic import when the manufacturer package is imported. In particular this means that to import just one instrument from one manufacturer you do not import all the drivers for the manufacturer instrument.

To achieve this, I3py replace the manufacturer package module by a custom one providing lazy import capabilities. For each manufacturer, the top level package should look like:

```
# -*- coding: utf-8 -*-
# -----------------------------------------------------------------------------
# Copyright 2018 by I3py Authors, see AUTHORS for more details.
#
# Distributed under the terms of the BSD license.
#
# The full license is in the file LICENCE, distributed with this software.
# -----------------------------------------------------------------------------
"""Package for the drivers of Itest instruments.

"""
import sys
from i3py.core.lazy_package import LazyPackage

DRIVERS = {'BN100': 'bn100.BN100'}

sys.modules[__name__] = LazyPackage(DRIVERS, __name__, __doc__, locals())
```

The *DRIVERS* dictionary contains a mapping between the name of the drivers that should be accessible and their import path (typically module_name.class_name). To make your driver visible simply add it to this dictionary. If you driver is defined in a sub-package and this package is itself lazy, your driver will be visible as soon as it is visible in the sub-package and that the sub-package is imported in the manufacturer package.

2.1.2 Features

Features are descriptors just like standard Python property. They live on the class and define what happens when one gets or sets the attribute associated with the feature on a class instance, that is to say when one writes:

```
a = d.myfeature
```

or:

```
d.myfeature = 1
```

The following sections will describe the different steps involved in the getting and setting process and how they can be customized using the different arguments it takes. Even more advanced customizations are possible and will be described in their own part *Advanced customization*.

Working principle

First we will describe the process involved in retrieving a feature value then switch to describing the setting of it.

Note: The first two arguments of a feature are always the getter and setter. If their value is set to None, the corresponding operation won't be possible for the feature. A feature is always deletable and deleting it corresponds to discarding the cached value if any exists.

Getting chain

First when getting a feature, we check if the instrument options allow to access it, and if not an AttributeError is raised. Next, we check whether or not its current value is known. If it is, the cached value is directly returned otherwise the system proceed with the retrieval sequence, in three steps as follows:

- `pre_get()`: This step is in charge to check that we can actually retrieve the value from the instrument. Some assertions about the instrument current state can for example be performed.
- `get()`: This step is tasked with the actual communication with the instrument. It should retrieve the value from the instrument and pass to the next step without performing any conversion. By default, it will call the `default_get_feature()` method defined on the class it belongs and pass it the value of the getter argument passed to the feature when it was created.
- `post_get()`: This step is tasked with converting the value obtained at the get step into a more user friendly representation than what was returned by the instrument. It can for example extract the meaningful part of the instrument response and turn it into the proper type, such as an integer or a float. It can also check for an error state on the instrument even so get operation should not cause any issue and such checks should be left to the setting.

The value coming out of the post_get step is cached and then returned to the user. If an error occurs during any of the step, if it is one of the ones listed in the retries_exceptions attribute of the driver the connection will be closed and re-opened and the operation attempted anew. Otherwise or if the re-opening fails too many times (more than specified in the retries argument of the feature), an `I3pyFailedGet` exceptions will be raised while still pointing to the original errors.

Setting chain

First when setting a feature, we check if the instrument options allow to access it, and if not an AttributeError is raised. Next, the value is checked against the cached value. If both values are found to be equal, the set is not performed as it would be useless. Otherwise, we proceed with the setting sequence, which, like the getting, happens in three steps:

- `pre_set()`: During this step, the state of the instrument can be checked and the value passed by the user validated and converted to a format appropriate to pass to the instrument.
- `set()`: This step is dedicated to actually communicating with the instrument to set the value. If the instrument returns any value that can be used to check that the operation went without issue, it should be returned so that it can be passed up to the next method. By default, it will call the `default_set_feature()` method defined on the class it belongs and pass it the value of the setter argument passed to the feature when it was created.
- `post_set()`: This step main goal is to check that the operation of setting the value went without trouble. By default, it simply calls the `default_check_operation()` on the parent class.

Once the value has been set and if no error occurred, the value specified by the user is cached. If an error occurs during any of the step, if it is one of the ones listed in the retries_exceptions attribute of the driver the connection will be closed and re-opened and the operation attempted anew. Otherwise or if the re-opening fails too many times (more than specified in the retries argument of the feature), an `I3pyFailedSet` exceptions will be raised while still pointing to the original errors.

Usual configurations

In addition to the ‘getter’ and ‘setter’ previously mentioned I3py features provides a number of often required checks, data extraction and data conversion utilities. The following list illustrates them:

- ‘options’: available on all `Feature` subclasses

  A “;” separated list of checks to perform on options values to determine if the Feature can be used. Options are defined using the `Options` feature. The test is performed a single time and then cached.

- ‘checks’: available on all `Feature` subclasses

  Similar to options, but can be used to check any value and is performed each time the feature is get or set.

- ‘extract’: available on all `Feature` subclasses

  A format string specifying how to extract the value of interest from the instrument response.

- ‘discard’: available on all `Feature` subclasses

  A list of features whose cache value should be discarded when the feature value is set. Alternatively a dict whose keys are ‘features’ and ‘limits’ can be used to also specify to discard some cached limits. One can access to the features and limits defined on the parent component using leading dots.

- ‘values’: available on `Str`, `Int` and `Float`

  A tuple of acceptable values for the feature.

- ‘mapping’: available on `Str`, `Int` and `Float`

  A mapping between user meaningful values and instrument meaningful ones.

- ‘limits’: available on `Int` and `Float`

  A 2-tuple, 3-tuple or str specifying the minimal and maximal values allowed and optionally the resolution that the feature can take. In the case of a str, the string specifies the named limit to use (see the following paragraph about defining limits).

- ‘aliases’: available on `Bool`

  A dictionary whose keys are True and False and whose values (list) specifies accepted aliases for True and False for setting.

Note: In many cases, the range of allowed values for a specific feature is not fixed but may be related to another feature value. To handle this case, I3py allows to define dynamic limits using the `limit()` decorator. The decorated method should return an instance of `IntLimitsValidator` or `FloatLimitsValidator` depending the kind of value this limit applies to.

Specialized features

- The `Alias` feature is a special feature allowing to delegate the actual work of getting/setting to another feature.
- The `Register` is a specialized feature which can be used to get and set the value of a binary register such as the ones commonly used by VISA based instrument. It will create a dedicated subclass of IntFlag and will handle the conversion. It takes two special arguments:
  - names: a list of names describing each bit in order (from least significant to most significant) or a dictionary mapping each name to the bit it describe. Those names should be valid python attribute names and ideally be all upper case.
  - length: the length of the register (8 by default but some instrument use 16 bits register).

- The `Options` is feature dedicated to the handling of hardware/firmware level options that cannot change while the instrument is running. It is expected to return a dictionary containing the values of the instrument options. To improve clarity the declaration of the feature should include the names of all the options to which it gives access along with hint about their possible values either as type or as a tuple of values. These information should be provided as a dictionary to the *names* argument.

Flexible getter/setter

In some cases, the command to use to get/set a Feature may depend on the state of the instrument. This use case can be handled by using a custom get/set method as described in *Advanced customization*. However as such cases can be quite common, I3py provides an alternative mechanism based on factory class to which the building of the get/set method can be deferred. Such factory classes should inherit from `AbstractGetSetFactory` and can be used for the getter/setter arguments of a feature.

The factories implemented in I3py can be found in *i3py.core.features.factories*.

2.1.3 Actions

Actions are the equivalent of features for methods. While features allows custom access to attributes, actions allow custom access and calling of methods. They allow in particular to specify checks or conversions on the methods arguments and return values.

The following sections will describe the different steps involved when calling an Action and how they can be customized using the different arguments it takes. Even more advanced customizations are possible and will be described in their own part *Advanced customization*.

Note: If some cases, an action may start an operation that the instrument will take a long time to process. In such a case it is best to return an `InstrJob` object that can be used at the appropriate time to wait for completion than to block the interface for a long time.

Working principle

When accessing an Action (d.action), we check first if the instrument options allow to access it, and if not an AttributeError is raised. Under normal circumstances, Python would return a bound method than we can next call. For actions, we return an `ActionCall` which takes in charge to run the three steps of the call:

- `pre_call()`: This step is in charge to check that we can actually perform the wrapped action. Some assertions about the instrument current state and argument values can for example be performed.
- `call()`: Call the wrapped method passing as argument the argument as returned by the pre-call step.
- `post_call()`: This step is tasked with converting the return values of the call and running some additional checks.

Usual configurations

In addition to the ‘getter’ and ‘setter’ previously mentioned I3py features provides a number of often required checks, data extraction and data conversion utilities. The following list illustrates them:

- ‘options’: available on all `BaseAction` subclasses

A “;” separated list of checks to perform on options values to determine if the Action can be used. Options are defined using the `Options` feature. The test is performed a single time and then cached.

- 'checks': available on `Action`

A ";" separated list of checks to perform each time the action is called. All the method arguments are available in the assertion execution namespace so one can access to the driver using self and to the arguments using their name (the signature of the wrapper is made to match the signature of the wrapped method).

- 'values': available on `Action`

A dictionary mapping the argument names to their allowed values (tuple). Arguments not listed in this dictionary are simply not validated.

- 'limits': available on `Action`

A dictionary mapping the argument names to their limits. Arguments not listed in this dictionary are simply not validated. Limits can be a 2-tuple, 3-tuple or str specifying the minimal and maximal values allowed and optionally the resolution that the feature can take. In the case of a str, the string specifies the named limit to use (see *Features* about defining limits).

Note: The `RegisterAction` is a specialized action which can be used to read the value of a binary register such as the ones commonly used by VISA based instrument. It will create a dedicated subclass of IntFlag and will handle the conversion. It takes two arguments:

- names: a list of names describing each bit in order (from least significant to most significant) or a dictionary mapping each name to the bit it describe.
- length: the length of the register (8 by default but some instrument use 16 bits register).

2.1.4 Subsystems and channels

Complex instruments usually have too many capabilities to fit reasonably in a single namespace, which is why SCPI commands usually define a hierarchy. Furthermore, either because the instrument is made of multiple parts or because the notion is built-in the instrument, another recurrent notion is the one of channel. Channels are usually identified by an id and share their capabilities. To handle those two cases I3py uses the notions of "subsystems" and "channels". As channels inherit a number of capabilities from subsystems, we will first describe them before moving on to the specificities of channels.

Subsystems

Subsystems act mainly as container and provide little capabilities by themselves. They do however allow to group options and checks for all their features and actions. The next following two sections focus on their declaration and on the working principle of options and checks.

Declaration

Subsystems can be declared in the body of a driver using the following syntax as already mentioned in *Writing a driver*.

```python
class MyDriver(VisaMessageDriver):
    """My driver with a subsystem.

    """
```

(continues on next page)

(continued from previous page)

```
    oscillator = subsystem()
    with oscillator as o:

        o.frequency = Float('OSC:FREQ?', 'OSC:FREQ {}')

        @o
        @Action()
        def is_sync(self):
            pass
```

Once created, the use of a context manager allows for the use of short names but also some additional magic and it should hence be used.

While convenient, this syntax can be cumbersome if one needs to declare nested subsystems/channels and as presented here would lead to large amount of code duplication for similar instruments. To allow the declaration of subsystems outside of a driver declaration, `subsystem` supports to be passed a list of base classes as first argument ('bases'). Those base classes do not have to be subsystems themselves (subclass of `AbstractSubSystem`) and if none of them are, a proper class will be added by the framework.

When subclassing a driver which has subsystems, one can modify the subsystems (adding/modifying actions/features) by simply redeclaring it with the same name and proceeding as for a new one. The framework will identify that the subsystem already exists and will use the version present on the base class as base class for the subsystem.

Note: In the case of multiple inheritance, if several of the driver base classes declare the same subsystem, the framework will use the one present on the first class of the mro (Method Resolution Order, ie the leftmost in the class creation). Other classes can be added as arguments of `subsystem`.

Options and checks

As mentioned in the introduction, subsystems can define tests (options and checks) that apply to all their features and actions. Those can be declared just like for features and actions by passing strings defining the options ('options' argument) and checks ('checks' argument).

The options will be tested when one try to access the subsystem from the driver:

```
ss = driver.subsystem
```

If the tests do not evaluate to true, an `AttributeError` will be raised mimicking a missing attribute. And as for all other options test the result will be cached. To implement this, the subsystem is accessed through a descriptor.

Note: By default, the framework uses `i3py.core.subsystem.SubSystemDescriptor` as descriptor to protect the access to a subsystem. You can specify an alternative descriptor using the 'descriptor' argument of subsystem. Alternative descriptor should inherit from `i3py.core.abstract.AbstractSubSystemDescriptor`.

Checks on the other hand are run each time a feature is accessed or an action is run. To achieve this, the framework customize the features/actions of the subsystem by adding an 'enabling' step to pre_get/set/call.

Note: When a inheriting a subsystem from a parent driver, the options and checks defined in the subsystem call are appended to the ones existing on the subsystem of the parent driver.

Features working in subsystems

In order for features to work in subsystems, subsystems implement: `default_get_feature()`, `default_set_feature()`, `default_check_operation()`. As a subsystem is nothing but a container, it simply propagate the call to its parent, without altering the arguments.

Channels

In several respects, channels are very similar to subsystems. Just as them, they follow mostly the same logic as far as subclassing is concerned and also support checks and options which work in the same way. The key difference between subsystems and channels is that where only one subsystem is instantiated per driver, multiple instances of a channel can be tied to the same driver. The following section will describe the differences between channels and subsystems.

Declaration

Channels are declared in the body of a driver using the following syntax as already hinted in *Writing a driver*. The key difference with a subsystem is that a way to identify the valid channels id is generally required as first argument.

```
class MyDriver(VisaMessageDriver):
    """My driver with a channel.

    """

    channels = channel((1, 2, 3),
                       aliases={1: ('A', 'a'), 2: 'B', 3: 'C'})
    with channels as c:

        c.frequency = Float('CH{ch_id}:FREQ?', 'OSC:FREQ {}')

        @c
        @Action()
        def is_sync(self):
            pass
```

The valid ids for channel can be declared as above as a tuple or list, which make sense when the number of channel is hardcoded in the device. Alternatively, one can pass the name of a method existing on the parent whose signature should be (self) -> Iterable.

In some cases, it may be handy to provide alternate names for channels for the sake of clarity. One can do so by declaring aliases. Aliases should be a dictionary whose keys match the ids of the channels and whose values are the allowed alternatives. Alternatives can be specified either as a simple value or as a list/tuple.

When subclassing a driver which has channels, if no channels ids are provided the method used on the parent driver will be inherited, and the aliases mapping will be updated with any new value provided (note that this will use the provided dict to update the inherited one such that duplicate keys will be overridden).

Note: As for subsystems one can specify base classes for a channel and the same inheritance rules apply.

Usage

As explained in the user guide, channel instances can be accessed using the following syntax:

```
driver.channels[ch_id]
```

where *ch_id* would 1, 2, 3 or any of their aliases in the previous case.

To achieve this and allow to check for options too, the channel machinery uses, like subsystems, a descriptor to protect the access to the object storing the channel instances, which we will refer to as the channel container. To make things clear, when writing:

```
c = driver.channels
```

c is the channel container returned by the descriptor. In addition to supporting subscription, the container is iterable and has the following attributes:

- available: list of the ids of the channels that can be accessed.
- aliases: mapping between the declared aliases and the matching channel id.

By default, the framework will use `i3py.core.base_channel.ChannelDescriptor` for the descriptor and `ChannelContainer` for the channel container. Just like for subsystems, it is possible to substitute to those classes custom ones using the *descriptor_type* and the *container_type* keyword arguments. The substitution classes should inherit from the proper abstract classes: `i3py.core.abstract.AbstractChannelDescriptor` and `AbstractChannelContainer` respectively.

Features working in channels

In order for features to work in channels, channels implement: `default_get_feature()`, `default_set_feature()`, `default_check_operation()`. In the case of the first two methods, a channel add its id under the keyword argument *ch_id* to the keyword arguments and propagate the call the parent driver. For the third method the call is simply forwarded on the parent.

The default behaviour is well fitted for VISA message based instruments when the channel id is part of the command as in this case things work out the box. The user simply has to indicate where to format the channel id, as illustrated in the above example. For instrument that requires first the channel to be selected, it is simply a matter of overriding the method to prepend the channel selection command.

2.1.5 Advanced customization

The mechanisms presented in the previous sections allow to handle a large number of situation occurring in real life instrument. However, in instruments, there are corner cases, and those are actually not so rare and require to be handled as gracefully as possible.

To handle those cases, I3py allows to customize the pre_get/set/call, get/set/call, post_get/set/call of features and actions either by replacing them by hand written function or by stacking on the existing behaviors custom functions. The following sections will present the mechanisms involved.

Defining custom handler

Custom handlers can be defined using the following syntax:

```
from i3py.core import customize

class MyDriver(VisaMessageDriver, IEEEIdentify):
    """My driver (supporting *IDN?) docstring.

```

(continues on next page)

(continued from previous page)

```
    """
    mode = Unicode('MODE?', 'MODE {}', values=('CW', 'PULSED'))

    @customize('mode', 'post_get')
    def _custom_mode_post_get(feat, driver, value)->Any:
        print(f'Read mode {value}')
        return value
```

Let examine in details how this works. First we import the `customize` decorator. `customize` is actually a class, which we first instantiate. We pass it the name of the feature/action on which it should apply, and the name of the method of this descriptor that should be customized. Because, we did not specify any additional argument the customization function will replace the existing one. This is the simplest way to use the customization mechanism, but it does not allow to combine existing mechanism with custom behavior. How to achieve this will be discussed in the next section.

Next, we use to decorate a function. **BE CAREFUL HERE**, even though we are in the body of a class, this function won't be bound as a method of this class, which is why we **DO NOT USE** self as first argument because it **WILL NOT BE** the first argument this function will take. With that in mind, one can notice that the signature of the function matches the signature of the descriptor method to customize. Actually, it should match the signature exactly (using the same argument names). Only self can be aliased to 'feat' when customizing a feature and 'action' for actions, for the sake of clarity.

Note: As the exact signature of the action, may be painful to emulate one can use instead *(action, driver, *args, **kwargs)* for pre_call and *(action, driver, result, *args, **kwargs)* for post_call.

As this mechanism is quite advanced, it is picky and one must be careful when using it. First, as already mentioned, the signature must be an exact match for input arguments but additionally one must be careful to return the expected value:

- None for `pre_get()` and `post_set()`
- a value for `get()` and `call()`
- the processed value for `post_get()`, `pre_set()` and `post_call()`
- a potential answer from the instrument for `set()`
- a tuple of argument and a dictionary of keyword arguments for `pre_call()`

Finally because of the way customization is handled, the function will not live as a method on the final class and hence one cannot simply modify the customization by simply overriding the method using the same name. One must explicitly requires a replacement.

Composing custom behavior and existing ones

When one needs an existing behavior (such as 'checks') and a custom on the same method, replacing the existing method would not be effective. To circumvent, this issue I3py allow to pass an additional behavior to customize to indicate that it should chain the call of the custom method and the existing one. To chain the calls, it will effectively replace the customized function by a `MethodComposer` instance which when called will call the chained functions.

The third argument of customize allow to specify how to perform the composition. It must be a tuple even when it contains a single argument and the possible values are the following:

- ('prepend',)
- ('add_before', existing_id)

- ('add_after', existing_id)
- ('append',)
- ('replace', existing_id)
- ('remove', existing_id)

The existing_id is a string allowing to identify the function with respect to which 'position' the customization. For built-in functionalities, it matches the keywords argument that was used to create it ('checks' for example).

Note: When a feature use multiple built-in mechanisms, those are composed using the same principle.

Note: Customization are also given an id. By default, it is simply custom, but one can specify a different value as the fourth argument to `customize`.

The way to chain the calls depends on the built-in function already present and the goal of the customization. For example, a custom conversion in a get may need to occur after the value was extracted from the instrument answer.

2.1.6 Standards

In order to improve inter-operability and allow, up to point, to replace one instrument with another equivalent one, it is crucial for both instrument interfaces to expose the same API to the user. I3py strives to achieve this kind of interoperability. Of course, it can never be perfect as commonly some instrument implement very specific behaviours not found in other. Furthermore, one must accept that generality may mean that the interface may not appear as simple as it could be because it takes into account possible variations in other instruments. One example is the one of the instrument output (voltage, microwave power, ...) in many cases one can find equivalent instruments with multiple outputs which is why the output of an instrument is often represented as a channel. This allows to simply change the output id when swapping two instruments. By convention, the id used for single output instruments is 0.

Trying to figure out, the "right" interface for a class of instrument is a tedious task that requires to consider for the initial design two or three instrument from different vendors. However once this work is done, implementing new drivers becomes straightforward. In addition, one can implement generic behaviours as part of a standard: the case of IEEE * commands is one example and the SCPI 'SYSTem:ERRor?' is another. Implementing those behaviours once in a standard allow to trivially support them in all instruments and limit code duplication.

To allow to use standards at any level of a hierarchy (top driver, subsytem, channel), all standards are implemented as simply inheriting `HasFeatures`. To use them they simply need to be added to the base classes of the component in which they belong.

2.2 Style guide

The uniformity of the coding style in a large project is of paramount importance to make maintenance easier. I3py follows closely PEP8 recommendations which can be found here (PEP8). One can automatically format code using the autopep8 tool. Some of those rules and some additional remarks are detailed below.

Contents

2.2.1 Header

All files part of I3py should start with the following header :

```
# -*- coding: utf-8 -*-
# -----------------------------------------------------------------------------
# Copyright 2016-2017 by I3py Authors, see AUTHORS for more details.
#
# Distributed under the terms of the BSD license.
#
# The full license is in the file LICENCE, distributed with this software.
# -----------------------------------------------------------------------------
```

New contributors should add their name to the AUTHORS file at the root of the project.

Immediately following this header one should find the module docstring.

2.2.2 Line length

PEP8 specifies that lines should at most 79 characters long and this rule is strictly enforced throughout I3py (in code and in comments). This makes the code much easier to read and on work on (one does not have to resize its editor window to accommodate long lines).

Backslashes should be used sparingly. To write an expression on multiple lines the preferred method should be to surround it with parenthesis.

Note: Long strings can use triple quotes or the following trick to avoid indentation issues :

```
msg = ('A very very long string, taking much more than a single line '
       'to write.')
```

The Python interpreter will automatically concatenate both strings when reading the file. Please that it will not insert any space or line feed (hence the space after 'line').

2.2.3 Docstrings

All functions, classes and methods should have a docstring (even private methods). I3py use the Numpy-style docstrings which are human readable.

2.2.4 Naming conventions

The naming conventions taken from PEP8 specifications are the following :

- local variables and functions should have all lowercase names and use '_' to separate different words. ex : my_variable
- class names should start with a capital letter and each new word should also start with one. ex : MyClass
- private variables or methods should start with a single '_'
- module constants should be in uppercase and use '_' to separate different words. ex : MY_CONSTANT

2.2.5 Import formatting

Imports should be at the top of the file (after the module docstring) save in special cases. They should be group as follow (each group separated from the following by a blank line) :

- special imports for Python 2/3 compatibility
- standard library imports
- third parties libraries imports
- relative imports

In each section the 'import x' stements should always come before the 'from a import b' statements.

2.2.6 Python version compatibility

I3py supports Python 3.5 and 3.6.

- *Driver machinery*

    You need to interface a new instrument, modify one or simply understand how the machinery works start here.

- tests/index

    You want to automatize the testing of your driver so that you can quickly know if your last update is safe, you should find the infos you need here.

- simulated/index

    You need a simulated instrument for testing, here are the explanations on how to write one.

Note: When writing code for I3py you should follow the project style guides described in *Style guide*.

CHAPTER 3

FAQS

- *User guide for I3py*

  How to set up I3py and use it.

- *Extending I3py*

  You need to add a new instrument to I3py, write some tests, add a simulated instrument, a functionality or simply understand what happens behind the scene you should start here.

- *FAQS*

  Some questions that might have occurred to others too.

- api_docs/index

  When all else fails, consult the API docs to find the answer you need. The API docs also include convenient links to the most definitive I3py documentation: the source.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search