IVI Driver Python Specification

Version Number	Date of Version	Version Notes
1.0	February 26, 2026	First approved version

Abstract

This specification contains the Python specific requirements for an IVI-Python driver, it is an IVI Language-Specific specification. This specification is to be used in conjunction with the IVI Driver Core Specification.

Authorship

This specification is developed by member companies of the IVI Foundation. Feedback is encouraged. To view the list of member vendors or provide feedback, please visit the IVI Foundation website at www.ivifoundation.org.

Warranty

Trademarks

Table of Contents

Overview of the IVI-Python Driver Language Specification

This specification contains the Python specific requirements for an IVI-Python driver, it is an IVI Language-Specific specification. Drivers that comply with this specification are also required to comply with the IVI Driver Core Specification.

This specification has several recommendations (identified by the use of the word *should* instead of *shall* in the requirement). These are included to provide a more consistent customer experience. However, in general, design decisions are left to the driver designer.

Substitutions

This specification uses paired angle brackets to indicate that the text between the brackets is not the actual text to use, but instead indicates the text that is used in place of the bracketed text. The IVI Driver Core Specification describes these substitutions.

Driver Identifier and Driver Class Name

This section specifies the substitutions for various forms of the *Driver Identifier* and the *Driver Class Name*.

The `<DriverIdentifier>` and its variations are used as identifiers within the driver that are unique to a particular driver. This section details the composition of the `<Driver Identifier>` and its variations. This section also defines the `<DriverClassName>` which is the top-level class instantiated by the driver client. The `<DriverClassName>` is only guaranteed to be unique within the scope of the `<DriverIdentifier>`.

The first token of the driver identifier shall indicate the *Instrument Manufacturer*, that is, the manufacturer of the instrument (or family of instruments) controlled by the driver. If the instrument manufacturer is not

registered in VPP-9, or if the driver supports multiple manufacturers' instruments the driver vendor may choose the characters, however the <DriverIdentifier> shall include the final token indicating the driver vendor.

The second token shall identify the instrument models supported, and any other driver identifying information the driver vendor chooses. If a vendor expects multiple versions of a driver to be used at once, the vendor must differentiate the identifiers by incorporating the driver version into the vendor-provided string. These additional characters shall not include underscores ('_').

If the *driver vendor* and the *instrument manufacturer* are different, a third token that indicates the <DriverVendor> shall be included in the <DriverIdentifier>. The <DriverVendor> may optionally be separated from the second token with an underscore ('_').

The token that identifies the <DriverVendor> and <InstrumentManufacturer> shall be a vendor abbreviation from *VPP-9*. Assigned identifiers can be found in the current version referenced at the IVI Foundation Specification Download page. This may be either the 2-character vendor abbreviation or the indefinite length vendor abbreviation. Vendors may register both identifiers with the IVI foundation for inclusion in *VPP-9* at no cost as described in *VPP-9*. Vendors are not permitted to duplicate identifiers that are already registered. Driver vendors are responsible for guaranteeing that the preceding part of the identifier is unique to the driver.

The characters that compose the <DriverIdentifier> shall remain the same throughout the driver. That is, the choice of *VPP-9* vendor abbreviation, and the optional use of the underscore separator must remain fixed for all uses of the <DriverIdentifier>.

In summary, the <DriverIdentifier> and <DriverClassName> are composed as follows (square brackets indicate the enclosed content is optional):

```
 <DriverVendor> ::= VPP-9 vendor abbreviation indicating the author of the driver
 <InstrumentManufacturer> ::= VPP-9 vendor abbreviation indicating the instrument
vendor
 <InstrumentModel> ::= Identifier indicating the instrument model or family of
instruments, as selected by the *Driver Vendor*. Shall not include the underscore
('_') character

 <DriverIdentifier> ::= <InstrumentManufacturer><InstrumentModel>[[_]
<DriverVendor>]
 <DriverClassName> ::= <InstrumentManufacturer><InstrumentModel>
```

Requirements:

- The <DriverVendor> may only be omitted if the *Driver Vendor* and *Instrument Manufacturer* are the same.
- The optional underscore ('_') preceding the <DriverVendor> may be included at the discretion of the *Driver Vendor*.

The case of the characters in the <DriverIdentifier> changes depending on the context of its use. This document uses the following conventions to specify the case when referring to the <DriverIdentifier>:

- <driver_identifier> refers to the driver identifier in lower case
- <DriverIdentifier> is used when the context does not require further clarification, or to indicate Pascal case. 2-character vendor abbreviations may be in upper case or Pascal case, at the vendor's discretion. If the optional separator is included in <driver_identifier> it is *included* in the <DriverIdentifier>.
- <DriverClassName> is always in Pascal case, however if a definite length (2-character) vendor abbreviation is used both characters may be upper case.

Examples:

In the following examples, the *Driver Vendor* and *Instrument Manufacturer* are the same:

```
For <DriverVendor> and <InstrumentManufacturer> the indefinite length form is
'Bask', and the definite length form is 'BI'.
For <InstrumentModel> the name is DMM (family of instruments).

The following are legal <DriverIdentifier>/<DriverClassName> pairs:

  # using the indefinite length names
    <DriverIdentifier> ::= BaskDmm
    <driver_identifier> ::= baskdmm
    <DriverClassName> ::= BaskDmm

  # using the definite length name
    <DriverIdentifier> ::= BIDmm
    <driver_identifier> ::= bidmm
    <DriverClassName> ::= BIDmm
```

In the following examples, the *Driver Vendor* and *Instrument Manufacturer* are different:

```
For <DriverVendor> the indefinite length form is 'Foo', and the definite length
form is 'FI'.
For <InstrumentManufacturer> the indefinite length is 'Bar' and the definite
length (2-character) form is 'BI'.
For <InstrumentModel> the model name is Tdr123A.

The following are legal <DriverIdentifier>/<DriverClassName> pairs:

# using the indefinite length names
  <DriverIdentifier> ::= BarTdr123AFoo
  <driver_identifier> ::= bartdr123afoo
  <DriverClassName> ::= BarTdr123A

# using the definite length names
  <DriverIdentifier> ::= BITdr123AFI
  <driver_identifier> ::= bitdr123afi
  <DriverClassName> ::= BiTdr123A
```

```
# using the definite length names mixed case
  <DriverIdentifier> ::= BiTdr123AFi
  <driver_identifier> ::= bitdr123afi
  <DriverClassName> ::= BiTdr123A

# using the definite length names mixed case with the optional underscore
  <DriverIdentifier> ::= BiTdr123A_Fi
  <driver_identifier> ::= bitdr123a_fi
  <DriverClassName> ::= BiTdr123A
```

IVI-Python Driver Architecture

This section describes how IVI-Python instrument drivers use Python. This section does not attempt to describe the technical features of Python, except where necessary to explain a particular IVI-Python feature. This section assumes that the reader is familiar with Python.

Style Guide

IVI-Python drivers shall comply with PEP-8 (*Style Guide for Python Code*).

Bitness

The IVI Python standard does not specify operating systems. Thus, there are no specific requirements around bitness. The IVI Compliance document shall thoroughly describe the capabilities of the driver and the environment(s) in which it is supported per the IVI Core specifications on driver identification.

Target Python Versions

IVI-Python drivers shall support at least one version of Python supported by the *Python Software Foundation* at the time of the driver's release. At the time of this writing, this website has details on the current supported versions: https://devguide.python.org/versions/. Drivers should support all active Python versions.

IVI-Python Naming

IVI-Python drivers shall follow the PEP-8 Python naming guidelines.

IVI-Python Packages

The IVI-Python driver shall be organized as a Python package, including an `__init__.py` file. Exactly one driver per distribution package shall be present.

IVI-Python Package Versioning

The package should use semantic versioning (semver).

IVI-Python Distribution Package Naming

The name of the package for the driver shall follow the Python naming guideline.

The name should be lowercase with all runs of the characters ., -, or _ replaced with a single - character. This can be implemented in Python with the `re` module:

```
import re

def normalize(name):
  return re.sub(r"[-_.]+", "-", name).lower()
```

The distribution package name shall be the same as the import package name except for the choice of separator. Distribution package names shall be all lower-case. Dashes and underscores are allowed.

Existing drivers with different name compositions are exempt from this rule.

IVI-Python Package Type-Hinting

The driver package shall provide complete type-hinting. An empty file named `py.typed` shall be included at the top level of the package.

IVI-Python Driver Classes

IVI-Python drivers are object-oriented. There shall be a root class that when instantiated provides the complete driver API. That class is instantiated for each resource to be controlled. The name of the class shall be `<DriverClassName>`. Note that the import package name and the root class do not collide because they have different casing and/or missing driver vendor.

For instance, in the following example the driver vendor and instrument manufacturer are the same, so the import statement would look like this:

```
from rssiggen42 import RSSigGen42
```

If the driver vendor and instrument manufacturer are not the same, the import statement looks like this:

```
from rssiggen42ni import RSSigGen42
```

IVI-Python Hierarchy

Modules within the driver may be named at the driver vendor's discretion. An IVI-Python driver shall organize the driver's API as a hierarchy of classes. Each of the interfaces is implemented by one of the driver's classes.

One of the classes provided by the driver shall be the IVI-specified driver utility class defined in IVI-Python Utility Interface

The root of the hierarchy shall be the main class `<DriverClassName>`.

The main class shall include properties that return references to child classes. A child class may in turn include properties that return references to its child classes, and so on. These *reference properties* may then be used to

navigate to any instrument functionality from the main class. The hierarchy may be arbitrarily deep.

The names of the reference properties should be the snake-case form of the class name it references.

Consider the following example code:

```
kt1234.cls2.cls3.measure()
```

`kt1234` is a reference to an instance of the main class. `kt1234` contains an interface reference property named `cls2` which returns an instance of the class `Cls2`. `Cls2` contains a reference property named `cls3`, which returns an instance of the class `Cls3`. `Cls3` contains the method `measure()`.

> **Observation:**
>
> > As the user types each of these names, code completion in the Python editor makes navigating the hierarchy easy. It displays a dropdown list of methods and properties in the corresponding class or interface. After typing `kt1234` followed by a period, a list of all the properties and methods in `kt1234` appears, allowing the user to select one. After selecting `cls2` and typing the period, a list of the methods and properties in `cls2` appears. After selecting `cls3` and typing the period, a list of the methods and properties in `cls3` appears, and the user can see and select `measure()`.

Repeated Capabilities

Repeated capabilities may be represented in two ways in IVI-Python drivers. Repeated capability instances may be specified by:

1. A method that selects the active instance (the *selector style*) for subsequent operations.
2. Selecting a particular instance from a collection (the *collection style*).

See the IVI Driver Core Specification Repeated Capabilities for details.

For IVI-Python drivers, collection style repeated capabilities are recommended.

Collection Style Repeated Capabilities and the Hierarchy

Collection style repeated capabilities consist of at least two classes. The first is the collection itself, and the second is the class of the object returned by the subscript operator (`[]`) of the collection. In the hierarchy, a reference property returns the collection object. Then the collection's subscript operator (`[]`) is used to return an item from the collection. Each item in the collection represents one instance of the repeated capability.

Collection style repeated capabilities may be indexed by a string, integer, enumerated type or other Python object.

Consider the following example code:

```
my_peak = kt1234.trace["B"].peak
```

`kt1234` is an instance of the main class. `kt1234` contains a reference property named `trace`, which returns a reference to the trace collection. The subscript operator (`["B"]`) selects the item identified by "B" from the collection and returns a reference to an object that uniquely represents the "B" trace. That class contains the property `peak`.

Collections may be implemented in a variety of ways:

- Many collections do not need to add or remove members after the driver is constructed. These can be implemented as Python read-only collections.
- Applications that need to dynamically add or remove methods can use appropriate types, such as `dict`.
- Developers may create custom collections.

Repeated Capability Reference Property Naming

Drivers should name the classes associated with Repeated Capability Reference Properties as described in this section.

In the following statements, `<RcName>` is the name of the repeated capability.

- Repeated capability collection classes should be named: `<RcName>Collection`
- The class returned by the collection's item operator should be named: `<RcName>`
- The class returned by the collection's item operator should include a property called *name*. The *name* property returns the physical repeated capability identifier defined by the specific driver for the repeated capability that corresponds to the index used to get it from the collection.

The reference property that returns the repeated capability collection should be a **plural word**, to hint to the user that the data type is a collection.

For example:

```
# 'channels' is a reference property with repeated capability
channels_collection = session.channels
session.channels['1'].range = 10.0
session.channels[Channels.CHANNEL_1].range = 10.0
session.channels[1].range = 10.0
```

In addition, to improve the user experience by utilizing code-completion, the drivers may implement method-like accessors with `Enum` and string parameter data types. In this case, the method accessor shall have the same name as the property, with the suffix `_item`:

```
session.channels_item('1').range = 10.0
session.channels_item(Channels.CHANNEL_1).range = 10.0
session.channels_item(1).range = 10.0
```

For example, consider a trigger repeated capability. Then `<RcName>` = `Trigger`, and the collection class is `TriggerCollection`. The code snippet below demonstrates the above recommendations:

```
from typing import Any

class Trigger:
  """Trigger functions of the instrument."""

  def __init__(self, name: str, value: int) -> None:
    self._name: str = name
    self._value: int = value
    self._level: float = 1.0

  @property
  def name(self) -> str:
    """Logical name representing the command value for the user."""
    return self._name

  @property
  def value(self) -> int:
    """Command value for the instrument."""
    return self._value

  @property
  def level(self) -> float:
    """Actual value of the Trigger property - trigger level."""
    return self._level

class TriggerCollection(dict[str, Trigger]):
  """Collection of the Trigger items."""

  def __init__(self) -> None:
    super().__init__()
    self["TriggerA"] = Trigger("TriggerA", 1)
    self["TriggerB"] = Trigger("TriggerB", 2)
    self["TriggerC"] = Trigger("TriggerC", 3)
```

IVI-Python Error Handling

All IVI-Python instrument drivers shall consistently use the standard Python exception mechanism to report errors. Neither return values nor *out* parameters shall be used to return error information.

> **Observation:**
>
> > The method `error_query()` can be used to read back errors from the instrument that may not be thrown as Python exceptions.

Documentation and Source Code

This specification does not have specific requirements on the format or distribution method of documentation and source code other than those called out in *IVI Driver Core Specification*. It requires that some drivers

provide source code and has detailed documentation requirements.

IVI Python drivers are permitted to distribute the required documentation online. The README file shall provide instructions to acquire the documentation.

Base IVI-Python API

This section gives a complete description of each constructor, method, and property required for an IVI-Python driver. The following table shows the mapping between the required base driver APIs described in the IVI Driver Core specification and the corresponding IVI-Python specific API described in this section.

Required Driver API Mapping Table

Required Driver API (IVI Driver Core)	**IVI-Python API**
Initialization	Driver Constructor
Driver Version	Property (r) `driver_version`
Driver Vendor	Property (r) `driver_vendor`
Error Query	Method `error_query()`
Instrument Manufacturer	Property (r) `instrument_manufacturer`
Instrument Model	Property (r) `instrument_model`
Query Instrument Status Enabled	Property (r/w) `query_instrument_status`
Reset	Method `reset()`
Simulate Enabled	Property (r) `simulate`
Supported Instrument Models	Property (r) `supported_instrument_models`

Additional Driver API

Besides the IVI Driver Core required API, the following additional methods shall be implemented for IVI-Python drivers:

- Method `error_query_all()` returns a collection of `ErrorQueryResult` objects that can also optionally implement a custom `__str__` method.
- Method `raise_on_device_error()` calls `error_query_all()` and raises an exception if any instrument errors were detected.

Constructor

In IVI-Python, the constructor provides the initialization functionality described in the IVI Driver Core Specification.

IVI-Python drivers shall implement a constructor with the following prototype, however it may have additional optional parameters after `options`:

`<DriverClassName>(resource_name: str, id_query: bool = True, reset: bool = False, options: dict | str | None = None)`

These required parameters are defined in the IVI Driver Core Specification. The following table shows their names and types for Python:

Inputs	Description	Data Type
`resource_name`	Resource Name	`str`
`id_query`	ID Query	`bool`
`reset`	Reset	`bool`

Notes:

- *IVI Driver Python* constructor is implemented on the class named `<DriverClassName>`.
- Simulation mode can be set via the optional items.

The `options` parameter shall permit the client to specify driver options (such as *simulation* or options as string).

Example for DriverIdentifier `MyPowerMeter`:

```
from pyvisa import ResourceManager
from pyvisa.resources import Resource

class MyPowerMeter:

  def __init__(self, resource_name: str, id_query: bool = True, reset: bool = False, options: dict | str | None = None) -> None:
    # Initialization of the Powermeter
    self.io: Resource = ResourceManager().open_resource(resource_name)
    self.id_query: bool = id_query
    self.reset: bool = reset
    self.options: dict = self._process_options(options)
```

For the `options` data type, Python `TypedDict` is recommended instead of a standard dictionary. At run-time, `TypedDict` is a standard `dict` type, but has the advantage of providing code-completion and type hinting in static analysis.

Example:

```
from typing import TypedDict

class Options(TypedDict, total=False):
  """Definition of the driver's Options items."""
  simulate: bool
  clear_status_on_init: str
  block_data_chunk: int
```

```
# User code usage
opt = Options()
opt['simulate'] = True
opt['block_data_chunk'] = 12
opt['simulate'] = 0 # static analysis shows an error on value type
opt['something'] = False # static analysis shows an error on key name
```

IVI-Python Utility Interface

IVI-Python drivers shall implement the class defined in this section. The driver shall provide a reference property to acquire the driver's instance of the utility class.

The reference property shall be named *ivi_utility*. The reference property shall be available on the root driver class. The driver developer is responsible for defining a class that inherits from `IviUtility` and is instantiated when the top driver class (that is the class named: `<DriverClassName>`) is instantiated.

```
from abc import ABC, abstractmethod
from typing import Any, List, Tuple, Collection

class ErrorQueryResult:

  def __init__(self, code: int, message: str) -> None:
    self._code = code
    self._message = message

  @property
  def code(self) -> int:
    return self._code

  @property
  def message(self) -> str:
    return self._message


class IviUtility(ABC):

  @property
  @abstractmethod
  def driver_version(self) -> str:
    pass

  @property
  @abstractmethod
  def driver_vendor(self) -> str:
    pass

  @property
  @abstractmethod
  def instrument_manufacturer(self) -> str:
    pass
```

```
    @property
    @abstractmethod
    def instrument_model(self) -> str:
      pass

    @property
    @abstractmethod
    def query_instrument_status_enabled(self) -> bool:
      pass

    @query_instrument_status_enabled.setter
    @abstractmethod
    def query_instrument_status_enabled(self, enabled: bool) -> None:
      pass

    @property
    @abstractmethod
    def simulation_enabled(self) -> bool:
      pass

    @abstractmethod
    def error_query(self) -> ErrorQueryResult | None:
      pass

    @abstractmethod
    def error_query_all(self) -> Collection[ErrorQueryResult]:
      return []

    @abstractmethod
    def raise_on_device_error(self) -> None:
      pass

    @abstractmethod
    def reset(self) -> None:
      pass

    @property
    @abstractmethod
    def supported_instrument_models(self) -> Tuple[str, ...]:
      pass
```

See IVI Driver Core Specification for general requirements.

Python Specific Notes:

- Drivers are permitted to allow setting the `simulate` property. However, if they do so, they shall properly manage the driver state when turning simulation on and off.

Direct IO Properties and Methods

Per the IVI Driver Core Specification, IVI Drivers for instruments that have an ASCII command set such as SCPI shall also provide an API for sending messages to and from the instrument over the ASCII command channel. This section specifies those properties and methods.

The reference property should be named *ivi_direct_io*. The reference property should be available on the root driver class.

```
from abc import ABC, abstractmethod
from typing import Any, List

class IviDirectIo(ABC):

  @property
  @abstractmethod
  def session(self) -> Any:
    pass

  @property
  @abstractmethod
  def io_timeout_ms(self) -> int:
    pass

  @io_timeout_ms.setter
  @abstractmethod
  def io_timeout_ms(self, timeout_ms: int) -> None:
    pass

  @abstractmethod
  def read_bytes(self) -> bytes:
    pass

  @abstractmethod
  def read_string(self) -> str:
    pass

  @abstractmethod
  def write_bytes(self, data: bytes) -> None:
    pass

  @abstractmethod
  def write_string(self, data: str) -> None:
    pass
```

Notes:

- The optional `session` property should return the underlying IO library.

Package Requirements

The following sections detail the package requirements.

Package Metadata

The instrument manufacturer and model(s) supported by the driver shall be mentioned in the keywords list. The forms of the manufacturer and the model(s) shall be the same as returned from the driver's API.

Below, is an example of the toml file content:

```
[project]
name = "vendorxy-specan"
version = "1.0"
requires-python = ">= 3.8"
authors = [ {name = "VendorXy"} ]
description = "This is a short description for the vendorxy-specan"
readme = {file = "README.md", content-type = "text/markdown"}
license = "MIT"
classifiers = [
  "Programming Language :: Python",
  "Topic :: Scientific/Engineering :: Instrument Drivers",
  "Topic :: Scientific/Engineering :: Instrument Drivers :: IVI Conformant"
]
dependencies = ["pyvisa"]
keywords = ["Manufacturer_XY", "SpecanModel_ABC"]

[project.urls]
Documentation = "https://readthedocs.org"
```

Package Contents

All IVI-Python driver packages shall include the following files:

- the driver
- The *README* file. The format of the file shall be Markdown (*.md) or reStructuredText (*.rst), where Markdown is recommended. The content of the file is specified in the IVI Driver Core Specification.
- the type hinting file (*py.typed*), at the top level of the package
- the documentation or directions for how to acquire it; directions are found in the *README* file
- If the source code is provided with this driver it may be in the package or the driver may provide instructions for how to acquire it in the *README* file. See the IVI Core Specification for details regarding when source code is required.
- the IVI Compliance document as specified in the IVI Core Specification

IVI-Python Driver Conformance

IVI-Python drivers are required to conform to all the rules in this document. They are also required to be registered on the IVI website.

Drivers that satisfy these requirements are *IVI-Python drivers* and may be referred to as such.

Registered conformant drivers are permitted to use the *IVI Conformant Logo*.

Driver Registration

Driver providers wishing to obtain and use the *IVI Conformant Logo* shall submit to the IVI Foundation the driver compliance document described in *IVI Driver Core Specification*, Section Driver Conformance along with driver information and a point of contact for the driver. The information shall be submitted to the IVI Foundation website. Complete upload instructions are available on the site. Driver vendors who submit compliance documents may use the IVI Conformant logo graphics.

The IVI Foundation may make some driver information available to the public for the purpose of promoting IVI drivers. All information is maintained in accordance with the IVI Privacy Policy, which is available on the IVI Foundation website.