PyVISA Documentation

Release 1.9.0

PyVISA Authors

Mar 30, 2018

Contents

PyVISA-sim is a backend for PyVISA. It allows you to simulate devices and therefore test your applications without having real instruments connected.

You can select the PyVISA-sim backend using **@sim** when instantiating the visa Resource Manager:

```
>>> import visa
>>> rm = visa.ResourceManager('@sim')
>>> rm.list_resources()
('ASRL1::INSTR')
>>> inst = rm.open_resource('ASRL1::INSTR', read_termination='\n')
>>> print(inst.query("?IDN"))
```

That's all! Except for **@sim**, the code is exactly what you would write in order to use the NI-VISA backend for PyVISA.

If you want to load your own file instead of the default, specify the path prepended to the @sim string:

```
>>> rm = visa.ResourceManager('your_mock_here.yaml@sim')
```

You can write your own simulators. See *Building your own simulated instruments* to find out how.

CHAPTER 1

Installation

Using pip:

```
pip install -U pyvisa-sim
```

You can report a problem or ask for features in the issue tracker.

1.1 User Guide

1.1.1 Building your own simulated instruments

PyVISA-sim provides some simulated instruments but the real cool thing is that it allows you to write your own in simple YAML files.

Here we will go through the structure of such a file, using the one provided with pyvisa-sim as an example. The first line you will find is the specification version:

```
spec: "1.1"
```

This allow us to introduce changes to the specification without breaking user's code and definition files. Hopefully we will not need to do that, but we like to be prepared. So, do not worry about this but make sure you include it.

The rest of the file can be divided in two sections: devices and resources. We will guide you through describing the Lantz Example Driver

devices

It is a dictionary that defines each device, its dialogues and properties. The keys of this dictionary are the device names which must be unique within this file. For example:

```
devices:
    HP33120A:
        <here goes the device definition>
    SR830:
        <here goes the device definition>
```

The device definition is a dictionary with the following keys:

eom

Specifies the end-of-message for each instrument type and resource class pair. For example:

```
eom:
  ASRL INSTR:
    q: "\r\n"
    r: "\n"
```

means that **ASRL INSTR** resource queries are expected to end in **rn** and responses end in **n**. The **q**, **r** pair is a common structure that will repeat in dialogues, getters and setters.

You can specify the eom for as many types as you like. The correct one will be selected when a device is assigned to a resource, as we will see later.

error

The error key specifies the default message to be given when a message is not understood or the user tries to set a property outside the right range. For example:

```
error: ERROR
```

This means that the word **ERROR** is returned.

If you want to further customize how your device handles errors, you can split the error types in two: **command_error** which is returned when fed an invalid command or an out of range command, or **query_error** which is returned when trying to read an empty buffer.

error:

response: command_error: null_response

status_register:

- q: "*ESR?" command_error: 32 query_error: 4

In addition to customizing how responses are generated you can specify a status register in which errors are tracked. Each element in the list specifies a single register so in the example above, if both a **command_error** and **query_error** are raised, then querying '*ESR?' will return '36'.

dialogues

This is one of the main concepts of PyVISA-sim. A dialogue is a query which may be followed by a response. The dialogues item is a list of elements, normally **q**, **r** pairs. Fore example:

```
dialogues:
  - q: "?IDN"
    r: "LSG Serial #1234"
```

If the response (**r**) is not provided, no response will be given by the device. Conversely, if **null_response** is provided for response (**r**), then no response will be given by the device as well.

You can have as many items as you want.

properties

This is the other important part of the device. Consider it as a dialogue with some memory. It is a dictionary. The key is the name of the property and the value is the property definition. For example:

```
properties:
  frequency:
    default: 100.0
    getter:
      q: "?FREQ"
      r: "{:.2f}"
    setter:
      q: "!FREQ {:.2f}"
      r: OK
    specs:
      min: 1
      max: 100000
      type: float
```

This says that there is a property called **frequency** with a default value of **100.0*.

To get the current frequency value you need to send **?FREQ** and the response will be formatted as **{:.2f}**. This is the PEP3101 formatting specification.

To set the frequency value you need to send **!FREQ** followed by a number formatted as **{:.2f}**. Again this is the PEP3101 formatting specification but used for parsing. If you want know more about it, take a look at the stringparser library. If setting the property was successful, the response will be **OK**. If there was an error, the response will be **ERROR** (the default). You can specify an error-specific error message for this setter as:

```
e: Some other error message.
```

Finally you can specify the specs of the property:

```
specs:
  min: 1
  max: 100000
  type: float
```

You can define the minimum (min) and maximum (max) values, and the type of the value (float, int, str). You can also specify the valid values, for example:

```
specs:
  valid: [1, 3, 5]
```

Notice that even if the type is a float, the communication is done with strings.

resources

It is a dictionary that binds resource names to device types. The keys of this dictionary are the resource names which must be unique within this file. For example:

```
resources:
    ASRL1::INSTR:
        device: device 1
    USB::0x1111::0x2222::0x1234::INSTR:
        device: device 1
```

Within each resource, the type is specified under the **device** key. The associated value (e.g **device 1**) must corresponds to one of the keys in the **devices** dictionary that is explained above. Notice that the same device type can be bound to different resource names, creating two different objects of the same type.

You can also bind a resource name to device defined in another file. Simply do:

```
ASRL3::INSTR:
    device: device 1
    filename: myfile.yaml
```

The path can specified in relation with the current file or in an absolute way.

If you want to use a file which is bundled with PyVISA-sim, just write:

```
ASRL3::INSTR:
    device: device 1
    filename: default.yaml
    bundled: true
```