[**pycomm3**](https://pycomm3.readthedocs.io/en/latest/#module-pycomm3)**- A Python Ethernet/IP library for communicating with Allen-Bradley PLCs.**

The development for this library is aimed at providing quick and convenient access for reading/writing data inside Allen-Bradley PLCs.

This library is supported on Python 3.6.1 and newer and contains 3 different Drivers

**LogixDriver**

This is the main driver for this library, it supports ControlLogix, CompactLogix, and Micro800 PLCs.

**SLCDriver**

This driver can be used for reading/writing data files in SLC 500 or MicroLogix PLCs. Some of the more advanced or automatic features are not supported.

**CIPDriver**

This is the base class for the other two drivers, it handles some common shared services. It can also be used for generic CIP messaging to other non-PLC devices.

**Documentation**

This Docs covers a basic overview of the library, full documentation can be found on [Read the Docs](https://pycomm3.readthedocs.io/en/latest/).

**LogixDriver**

### Highlighted Features

* simple API, only 1 read method and 1 write method for tags.
  + does not require using different methods for different data types
  + requires the tag name only, no other information required from the user
  + automatically manages request/response size to pack as many requests into a single packet
  + automatically handles fragmented requests for large tags that can't fit in a single packet
  + both support full structure reading/writing (UDTs, AOIs, etc)
    - for read the Tag.value will be a dict of {attribute: value}
    - for write the value should be a sequence (list, tuple) of values for each attribute, nesting as needed
      * does not do partial writes, the value must match the complete structure
      * not recommended for builtin type (TIMER, CONTROL, COUNTER, etc)
* generic\_message for extra functionality not directly implemented
  + working similar to the MSG instruction in Logix, arguments similar to the MESSAGE properties
  + tested getting/setting drive parameters (see under examples in docs)
  + used internally to implement some of the other methods (get/set\_plc\_time, forward open/close, etc)
* simplified data types
  + strings use normal Python str objects, does not require reading/writing of the LEN and DATA attributes
  + BOOL arrays use normal Python bool objects, does not require complicated bit shifting of the DWORD value
* uploads the tag list and data type definitions from the PLC
  + no requirement for user to determine tags available (like from an L5X export)
  + controller-scoped tags by default, program-scoped tags are optional
  + definitions are required for read/write methods
* automatically enables/disables different features based on the target PLC
  + Extended Forward Open (EN2T or newer and v20+)
  + Symbol Instance Addressing (Logix v21+)
  + detection of Micro800 and disables unsupported features (CIP Path, Ex. Forward Open, Instance Addressing, etc)

### Basic Usage

Connect to a PLC and get some basic information about it. The path argument is the only one required, and it has 3 forms:

* IP Address Only (10.20.30.100) - Use if PLC is in slot 0 or if connecting to CompactLogix
* IP Address/Slot (10.20.30.100/1) - Use if PLC is not in slot 0
* CIP Routing Path (10.20.30.100/backplane/3/enet/10.20.40.100/backplane/0) - Use for more complex routing
  + first 2 examples will be replaced with the full path automatically
  + enet/backplane (or bp) are for port selection

### Example

from pycomm3 import LogixDriver

with LogixDriver('10.20.30.100/1') as plc:

print(plc)

# OUTPUT:

# Program Name: PLCA, Device: 1756-L83E/B, Revision: 28.13

print(plc.info)

# OUTPUT:

# {'vendor': 'Rockwell Automation/Allen-Bradley', 'product\_type': 'Programmable

Logic Controller', 'product\_code': 166, 'version\_major': 28, 'version\_minor':

13, 'revision': '28.13', 'serial': 'FFFFFFFF','device\_type': '1756-L83E/B',

'keyswitch': 'REMOTE RUN', 'name': 'PLCA'}

By default, when creating the LogixDriver object, it will open a connection to the plc, read the program name, get the controller info, and get all the controller scoped tags. By reading the tag list first, this allows us to cache all the tag type/structure information, including the instance ids for all the tags. This information allows the read/write methods to require only the tag name. If your project will require program-scoped tags, be sure to set the init\_program\_tags kwarg. By default, only the controller-scoped tags will be retrieved and cached.

**Reading/Writing Tags**

Reading or writing tags is as simple as calling the read and write methods. Both methods accept any number of tags, and will automatically pack multiple tags into a Multiple Service Packet Service (0x0A) while making sure to stay below the connection size. If there is a tag value that cannot fit within the request/reply packet, it will automatically handle that tag independently using the Read Tag Fragmented (0x52) or Write Tag Fragmented (0x53) requests.

Both methods will return Tag objects to reflect the success or failure of the operation.

### Example

with LogixDriver('10.20.30.100') as plc:

plc.read('tag1', 'tag2', 'tag3') # read multiple tags

plc.read('array{10}') # read 10 elements starting at 0 from an array

plc.read('array[5]{20}) # read 20 elements starting at elements 5 from an array

plc.read('string\_tag') # read a string tag and get a string

# writes require a sequence of tuples of [(tag name, value), ... ]

plc.write(('tag1', 0), ('tag2', 1), ('tag3', 2)) # write multiple tags

plc.write(('array{5}', [1, 2, 3, 4, 5])) # write 5 elements to an array starting

at the 0 element

plc.write(('array[10]{5}', [1, 2, 3, 4, 5])) # write 5 elements to an array

starting at element 10

plc.write(('string\_tag', 'Hello World!')) # write to a string tag with a string

plc.write(('string\_array[2]{5}', 'Write an array of strings'.split())) # write an

array of 5 strings starting at element 2

Note:

Tag names for both read and write are case-sensitive and are required to be the same as they are named in the controller.

**Reading/Writing Tags**

Tag definitions are uploaded from the controller automatically when connecting. This allows the read/writing methods to work. These definitions contain information like instance ids and structure size and composition. The tag definitions are accessible from the tags attribute. The tags property is a dict of {tag name: definition}.

Definitions for structures are accessible from the data\_types attribute. These include things like User-Defined Data Types (UDT), Add-On Instructions (AOI), strings, and pre-defined types (TIMER, COUNTER, etc). For structure tags (tag['tag\_type'] == 'struct'), the data type definition will be stored in the data\_type attribute. ('atomic' tags will only have a string with their data type name: 'DINT', 'REAL', ...).