pyA2L Documentation

Release 1.0.342

Christoph Schueler

Jun 24, 2026

CONTENTS:

1 Readme **3**
- 1.1 Contents . . . 3
- 1.2 About ASAM MCD-2 MC (ASAP2) . . . 4
- 1.3 What pyA2L offers . . . 4
- 1.4 Why pyA2L? . . . 4
- 1.5 Design highlights . . . 4
- 1.6 Installation . . . 5
- 1.7 Getting Started (Quickstart) . . . 5
- 1.8 Tips . . . 7
- 1.9 Examples . . . 8
- 1.10 Create API and coverage parity . . . 8
- 1.11 Command-line usage . . . 9
- 1.12 Compatibility . . . 9
- 1.13 Project links . . . 9
- 1.14 Contributing . . . 9
- 1.15 Code of Conduct . . . 9
- 1.16 Changelog / Release notes . . . 10

2 Getting Started with pyA2L **11**
- 2.1 Import an .a2l file to database . . . 11
- 2.2 Re-importing an existing file . . . 11
- 2.3 Open an existing .a2ldb database . . . 11
- 2.4 Running a first database query . . . 12
- 2.5 Using the Inspect API . . . 12
- 2.6 Working with IF_DATA . . . 13
- 2.7 Validate and export . . . 14
- 2.8 Next steps . . . 14
- 2.9 Migrating from `pya2l.DB` . . . 14

3 Installation **17**
- 3.1 Prerequisites . . . 17
- 3.2 Install from PyPI . . . 17
- 3.3 Install from source (editable) . . . 17

4 Building pyA2L (from source) **19**
- 4.1 Prerequisites . . . 19
- 4.2 One-liner build and install (recommended) . . . 19
- 4.3 Development install . . . 19
- 4.4 Building distribution artifacts . . . 19
- 4.5 Building the documentation . . . 20

5 Configuration **21**
5.1 Import options . . . 21
5.2 Export options . . . 22
5.3 Environment variables . . . 22
5.4 CLI options (a2ldb-imex) . . . 22
5.5 Database pragmas . . . 23
5.6 Logging . . . 23

6 Tutorial **25**
6.1 Basic Usage . . . 25
6.2 Working with Modules . . . 26
6.3 Advanced Features . . . 28
6.4 Working with Variant Coding . . . 32
6.5 Common Use Cases . . . 32
6.6 Best Practices . . . 33
6.7 Conclusion . . . 34

7 Working with IF_DATA **35**
7.1 The IfData class . . . 35
7.2 Quick start . . . 36
7.3 Parsed data vs. raw data . . . 36
7.4 Using the flatmap . . . 36
7.5 Accessing IF_DATA across all entity types . . . 37
7.6 Practical examples . . . 38
7.7 Manual parsing with IfDataParser . . . 41
7.8 Empty IF_DATA and edge cases . . . 42
7.9 Best practices . . . 42

8 API Reference **45**
8.1 Session Management . . . 46
8.2 Inspect API . . . 46
8.3 Create API . . . 54
8.4 Validate API . . . 61
8.5 Low-Level ORM Access . . . 63

9 HOW-TOs **65**
9.1 Import once, reuse the database . . . 65
9.2 Export back to A2L or JSON . . . 65
9.3 CLI import/export (`a2ldb-imex`) . . . 67
9.4 Concurrent access and export safety . . . 69
9.5 Dump measurements to Excel . . . 70
9.6 Handle encodings and quiet mode . . . 70
9.7 Creating A2L content programmatically . . . 71
9.8 Performance & Best Practices . . . 76

10 Frequently Asked Questions **81**
10.1 Where to start? . . . 81
10.2 While importing my A2L file I'm getting strange Unicode decode errors, what can I do? . . . 81
10.3 My A2L file includes tons of files… Do I have to copy all of them to my current working directory? . . . 82
10.4 How do I work with IF_DATA sections in A2L files? . . . 82
10.5 How do I create new A2L elements programmatically? . . . 83
10.6 How do I filter query results when working with A2L elements? . . . 84
10.7 Can SYMBOL_LINK have a missing offset? . . . 84
10.8 What performance can I expect for large A2L files? . . . 85
10.9 Does the exporter preserve all A2L attributes during roundtrip? . . . 86

10.10 Any missing questions and answers? 86

11 pya2l **87**
11.1 pya2l package 87

12 Indices and tables **89**

pyA2L is a Python toolkit for working with ASAM MCD-2 MC (ASAP2) A2L files. The library parses A2L into a SQLite-backed model, provides inspection and validation utilities, and enables automation around ECU measurement and calibration metadata.

For background on the ASAP2 standard, see: https://www.asam.net/standards/detail/mcd-2-mc/wiki/

CHAPTER

ONE

README

pyA2L is an ASAM MCD-2MC processing library written in Python.

If you work with ECUs, ASAP2/A2L is the contract that describes what and how to measure or calibrate. pyA2L helps you parse once, inspect and validate programmatically, automate checks, and export back when needed — all from Python.

1.1 Contents

- About ASAM MCD-2 MC (ASAP2)
- What pyA2L offers
- Installation
- Getting Started (Quickstart)
- Command-line usage
- Tips
- Examples
- Compatibility

- Project links
- Contributing
- Code of Conduct
- License
- Changelog / Release notes
- Acknowledgements

1.2 About ASAM MCD-2 MC (ASAP2)

ASAM MCD-2 MC (ASAP2) is the ASAM standard that defines the A2L description format for ECU measurement signals and calibration parameters. In practice, the A2L acts as a contract between ECU software and tools so different vendors can consistently locate data in memory and convert raw values into physical engineering units. Runtime transport (e.g., CCP/XCP) is out of scope of the standard.

For an authoritative overview of the standard, see the ASAM page: https://www.asam.net/standards/detail/mcd-2-mc/

1.3 What pyA2L offers

- Parse .a2l files and persist them as compact, queryable SQLite databases (.a2ldb) to avoid repeated parse costs.
- Programmatic access to ASAP2 entities via SQLAlchemy ORM models (MODULE, MEASUREMENT, CHARACTERISTIC, AXIS_DESCR, RECORD_LAYOUT, COMPU_METHOD/COMPU_TAB, UNIT, FUNCTION, GROUP, VARIANT_CODING, etc.).
- Rich inspection helpers in pya2l.api.inspect (e.g., Characteristic, Measurement, AxisDescr, ModPar, ModCommon) to compute shapes, axis info, allocated memory, conversions, and more.
- Creator API in pya2l.api.create to programmatically build or augment A2L content (PROJECT, MODULE, MEASUREMENT, CHARACTERISTIC, COMPU_METHOD, AXIS_PTS, RECORD_LAYOUT, GROUP, FUNCTION, etc.).
- Validation utilities (pya2l.api.validate) to check common ASAP2 rules and project-specific consistency.
- Export .a2ldb content back to A2L text or JSON when needed.
- Building blocks for automation: reporting, quality gates, CI checks, and integration with CCP/XCP workflows.

Supported ASAP2 version: 1.7.1

1.4 Why pyA2L?

- Parse once, query fast: Avoid repeated parser runs by working from SQLite.
- Powerful model: Use SQLAlchemy ORM to navigate ASAP2 entities naturally.
- Beyond parsing: Inspect derived properties, validate consistency, and export back to A2L.
- Automate: Integrate with CI to enforce quality gates on A2L content.

1.5 Design highlights

- **SQLite-backed storage** (.a2ldb) with SQLAlchemy models
- **C++ parser** (ANTLR4-based) for fast parsing (2.5 MB/s)

- **Adaptive flush strategy** automatically tunes performance (10% speedup for large files)
- **High-level inspection helpers** in `pya2l.api.inspect`
- **Creator API** in `pya2l.api.create` for programmatic A2L generation
- **Validator framework** in `pya2l.api.validate` yielding structured diagnostics
- **Export to A2L text or JSON** format with complete roundtrip fidelity
- **Optional CLI** (`a2ldb-imex`) for import/export tasks
- **Concurrent access** via SQLite WAL mode (multiple readers during export)

Learn more about the standard at the ASAM website: https://www.asam.net/standards/detail/mcd-2-mc/wiki/

1.6 Installation

- Via `pip`: `shell $ pip install pya2ldb` **IMPORTANT**: Package-name is `pya2ldb` **NOT** `pya2l`!!!
- From Github:
  - Clone / fork / download pyA2Ldb repository.

1.7 Getting Started (Quickstart)

Parse once, work from SQLite thereafter.

Import a .a2l file and persist it as .a2ldb (SQLite):

```
from pya2l import import_a2l

session = import_a2l(
    "ASAP2_Demo_V161.a2l",
    # Optional:
    # encoding="utf-8",        # default is latin-1
    # progress_bar=False,      # silence the progress meter
    # loglevel="ERROR",        # also suppresses progress
    # force_overwrite=True,    # overwrite existing .a2ldb without prompting
)
# Creates ASAP2_Demo_V161.a2ldb in the working directory
```

- Open an existing .a2ldb without re-parsing:

```
from pya2l import open_existing

session = open_existing("ASAP2_Demo_V161")  # extension .a2ldb is implied
```

Query with SQLAlchemy ORM - List all measurements ordered by name with address and data type:

```
from pya2l import open_existing
import pya2l.model as model

session = open_existing("ASAP2_Demo_V161")
measurements = (
    session.query(model.Measurement)
    .order_by(model.Measurement.name)
```

(continues on next page)

(continued from previous page)

```
    .all()
)
for m in measurements:
    print(f"{m.name:48} {m.datatype:12} 0x{m.ecu_address.address:08x}")
```

High-level inspection helpers - Use convenience wrappers from pya2l.api.inspect to access derived info:

```
from pya2l import open_existing
from pya2l.api.inspect import Characteristic, Measurement, AxisDescr

session = open_existing("ASAP2_Demo_V161")
ch = Characteristic(session, "ASAM.C.MAP.UBYTE.IDENTICAL")
print("shape:", ch.dim().shape)
print("element size:", ch.fnc_element_size(), "bytes")
print("num axes:", ch.num_axes())

me = Measurement(session, "ASAM.M.SCALAR.UBYTE.IDENTICAL")
print("is virtual:", me.is_virtual())

axis = ch.axisDescription("X")
print("axis info:", axis.axisDescription("X"))
```

Create A2L content programmatically - Use the Creator API to build or augment A2L databases:

```
from pya2l import open_create
from pya2l.api.create import (
    ProjectCreator, ModuleCreator, MeasurementCreator,
    CharacteristicCreator, CompuMethodCreator
)

session = open_create("MyProject.a2ldb")

# Create project and module
pc = ProjectCreator(session)
project = pc.create_project("MyProject", "Demo ECU Project")

mc = ModuleCreator(session)
module = mc.create_module("MyModule", "Demo Module", project=project)

# Add a conversion method
cmc = CompuMethodCreator(session)
cm = cmc.create_compu_method(
    "CM_Voltage", "Voltage conversion", "LINEAR",
    "%6.3", "V", module_name="MyModule"
)
cmc.add_coeffs_linear(cm, offset=0.0, factor=0.01)  # y = 0.01*x + 0

# Add a measurement
mec = MeasurementCreator(session)
meas = mec.create_measurement(
    "BatteryVoltage", "Battery voltage in Volts",
    "UWORD", "CM_Voltage", resolution=1, accuracy=0.1,
```

(continues on next page)

(continued from previous page)

```
    lower_limit=0.0, upper_limit=20.0,
    module_name="MyModule"
)
mec.add_ecu_address(meas, 0x1000)

# Add a characteristic (calibration parameter)
cc = CharacteristicCreator(session)
char = cc.create_characteristic(
    "InjectionMap", "Fuel injection map",
    "MAP", 0x2000, "RL_InjMap", 0.0, "CM_Voltage",
    0.0, 100.0, module_name="MyModule"
)

mc.commit()
db.close()
```

Validate your database

```
from pya2l import open_existing
from pya2l.api.validate import Validator

session = open_existing("ASAP2_Demo_V161")
vd = Validator(session)
for msg in vd():  # iterate diagnostics
    # msg has fields: type (Level), category (Category), diag_code (Diagnostics), text␣
↪(str)
    print(msg.type.name, msg.category.name, msg.diag_code.name, "-", msg.text)
```

Export back to A2L or JSON

```
from pya2l import export_a2l

# Export to A2L text format
export_a2l("ASAP2_Demo_V161", "exported.a2l")

# Or export to JSON for further processing
from pya2l.imex.json_exporter import export_json
export_json("ASAP2_Demo_V161.a2ldb", "exported.json")
```

1.8 Tips

- Default import encoding is latin-1; override `encoding=` if your file differs.
- Silence the progress meter via `progress_bar=False` or `loglevel="ERROR"`.
- Python package name is `pya2ldb` (not `pya2l`).
- See howto for Excel export and other short recipes.

1.9 Examples

- See `pya2l/examples` for sample A2L files and scripts.
- The Sphinx docs contain a fuller tutorial and HOWTO guides.

1.10 Create API and coverage parity

pyA2L offers a Creator API in pya2l.api.create to programmatically build or augment A2L content. The project's goal is coverage parity: everything you can query via pya2l.api.inspect is intended to be creatable via pya2l.api.create.

Example: creating common entities

```
from pya2l import open_create
from pya2l.api.create import ModuleCreator
from pya2l.api.inspect import Module

# Open or create a database
session = open_create("MyProject.a2l")  # or .a2ldb

mc = ModuleCreator(session)
# Create a module
mod = mc.create_module("DEMO", "Demo ECU module")

# Units and conversions
temp_unit = mc.add_unit(mod, name="degC", long_identifier="Celsius",
                        display="°C", type_str="TEMPERATURE")
ct = mc.add_compu_tab(mod, name="TAB_NOINTP_DEMO", long_identifier="Demo Tab",
                      conversion_type="TAB_NOINTP",
                      pairs=[(0, 0.0), (100, 1.0)], default_numeric=0.0)

# Frames and transformers
fr = mc.add_frame(mod, name="FRAME1", long_identifier="Demo frame",
                  scaling_unit=1, rate=10, measurements=["ENGINE_SPEED"])
tr = mc.add_transformer(mod, name="TR1", version="1.0",
                        executable32="tr32.dll", executable64="tr64.dll",
                        timeout=1000, trigger="ON_CHANGE", inverse="NONE",
                        in_objects=["ENGINE_SPEED"], out_objects=["SPEED_PHYS"])

# Typedefs and instances
ts = mc.add_typedef_structure(mod, name="TSig", long_identifier="Signal",
                              size=8)
mc.add_structure_component(ts, name="raw", typedefName="UWORD", addressOffset=0)
inst = mc.add_instance(mod, name="S1", long_identifier="Inst of TSig",
                       type_name="TSig", address=0x1000)

# Verify with inspect helpers
mi = Module(session)
print("#frames:", len(list(mi.frame.query())))
print("#compu tabs:", len(list(mi.compu_tab.query())))
```

See pya2l/examples/create_quickstart.py for a more complete example.

1.11 Command-line usage

A small CLI is provided as a console script named `a2ldb-imex`:

```
# Show version (exit 0)
$ a2ldb-imex -V

# Import an A2L (creates .a2ldb next to the input or in CWD with -L)
$ a2ldb-imex -i path/to/file.a2l

# Import with explicit encoding and create DB in current directory
$ a2ldb-imex -i path/to/file.a2l -E latin-1 -L

# Overwrite existing .a2ldb without prompting
$ a2ldb-imex -i path/to/file.a2l -f

# Suppress all output (useful in scripts/CI)
$ a2ldb-imex -i path/to/file.a2l -q

# Export an .a2ldb back to A2L text (stdout by default or -o file)
$ a2ldb-imex -e path/to/file.a2ldb -o exported.a2l
```

1.12 Compatibility

- Python: 3.10 – 3.14
- Platforms: Prebuilt wheels are published for selected platforms. From source, Windows/macOS are supported; Linux may require building native extensions.

1.13 Project links

- Source code: https://github.com/christoph2/pyA2L
- Issue tracker: https://github.com/christoph2/pyA2L/issues
- PyPI: https://pypi.org/project/pya2ldb/
- Documentation
- PDF Manual

1.14 Contributing

Contributions are welcome! Please open an issue to discuss significant changes before submitting a PR. See the existing tests under `pya2l/tests` and examples under `pya2l/examples` to get started. Contributors should use pre-commit to run formatting and lint checks before committing; see https://pre-commit.com/ for installation and usage.

1.15 Code of Conduct

This project follows a Code of Conduct to foster an open and welcoming community. Please read and abide by it when interacting in issues, discussions, and pull requests.

See CODE_OF_CONDUCT for full details.

1.16 Changelog / Release notes

See GitHub Releases: https://github.com/christoph2/pyA2L/releases

CHAPTER

TWO

GETTING STARTED WITH PYA2L

You'll find the example code here.

2.1 Import an .a2l file to database

```
from pya2l import import_a2l

session = import_a2l(
    "ASAP2_Demo_V161.a2l",
    # encoding="latin-1" is default; override if your file differs
    # progress_bar=False or loglevel="ERROR" to silence progress
)
```

If nothing went wrong, your working directory now contains a file named *ASAP2_Demo_V161.a2ldb*, which is simply a Sqlite3 database file.

Unlike other ASAP2 toolkits, you are not required to parse your *.a2l* files over and over again, which can be quite expensive.

2.2 Re-importing an existing file

By default, re-importing a file whose *.a2ldb* already exists triggers a confirmation prompt instead of raising an `OSError`. To suppress the prompt and overwrite unconditionally, pass `force_overwrite=True`:

```
from pya2l import import_a2l

session = import_a2l("ASAP2_Demo_V161.a2l", force_overwrite=True)
```

The CLI flag `-f` / `--force-overwrite` provides the same behaviour:

```
a2ldb-imex -i ASAP2_Demo_V161.a2l -f
```

2.3 Open an existing .a2ldb database

```
from pya2l import open_existing

session = open_existing("ASAP2_Demo_V161")   # No need to specify extension .a2ldb
```

You may have noticed, that in both cases the return value is stored in an object named *session*:

Enter SQLAlchemy!

SQLAlchemy offers, amongst other things, a powerful expression language.

2.4 Running a first database query

```
from pya2l import open_existing
import pya2l.model as model

session = open_existing("ASAP2_Demo_V161")
measurements = session.query(model.Measurement).order_by(model.Measurement.name).all()
for m in measurements:
    print(f"{m.name:48} {m.datatype:12} 0x{m.ecu_address.address:08x}")
```

Yields the following output:

```
ASAM.M.ARRAY_SIZE_16.UBYTE.IDENTICAL                UBYTE        0x00013a30
ASAM.M.MATRIX_DIM_16_1_1.UBYTE.IDENTICAL            UBYTE        0x00013a30
ASAM.M.MATRIX_DIM_8_2_1.UBYTE.IDENTICAL             UBYTE        0x00013a30
ASAM.M.MATRIX_DIM_8_4_2.UBYTE.IDENTICAL             UBYTE        0x00013a30
ASAM.M.SCALAR.FLOAT32.IDENTICAL                     FLOAT32_IEEE 0x00013a10
ASAM.M.SCALAR.FLOAT64.IDENTICAL                     FLOAT64_IEEE 0x00013a14
ASAM.M.SCALAR.SBYTE.IDENTICAL                       SBYTE        0x00013a01
ASAM.M.SCALAR.SBYTE.LINEAR_MUL_2                    SBYTE        0x00013a01
ASAM.M.SCALAR.SLONG.IDENTICAL                       SLONG        0x00013a0c
ASAM.M.SCALAR.SWORD.IDENTICAL                       SWORD        0x00013a04
ASAM.M.SCALAR.UBYTE.FORM_X_PLUS_4                   UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.IDENTICAL                       UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.TAB_INTP_DEFAULT_VALUE          UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.TAB_INTP_NO_DEFAULT_VALUE       UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.TAB_NOINTP_DEFAULT_VALUE        UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.TAB_NOINTP_NO_DEFAULT_VALUE     UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.TAB_VERB_DEFAULT_VALUE          UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.TAB_VERB_NO_DEFAULT_VALUE       UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.VTAB_RANGE_DEFAULT_VALUE        UBYTE        0x00013a00
ASAM.M.SCALAR.UBYTE.VTAB_RANGE_NO_DEFAULT_VALUE     UBYTE        0x00013a00
ASAM.M.SCALAR.ULONG.IDENTICAL                       ULONG        0x00013a08
ASAM.M.SCALAR.UWORD.IDENTICAL                       UWORD        0x00013a02
ASAM.M.SCALAR.UWORD.IDENTICAL.BITMASK_0008          UWORD        0x00013a20
ASAM.M.SCALAR.UWORD.IDENTICAL.BITMASK_0FF0          UWORD        0x00013a20
ASAM.M.VIRTUAL.SCALAR.SWORD.PHYSICAL                SWORD        0x00000000
```

The classes describing an *.a2ldb* database live in pya2l.model, they are required to query, modify, and add model instances.

2.5 Using the Inspect API

The inspect API provides high-level, read-only wrappers with automatic conversion and caching:

```
from pya2l import open_existing
from pya2l.api.inspect import Project, Measurement, Characteristic

session = open_existing("ASAP2_Demo_V161")
```

(continues on next page)

(continued from previous page)

```
# Navigate the project hierarchy
project = Project(session)
module = project.module[0]
print(f"Project: {project.name}")
print(f"Module:  {module.name}")

# Get a measurement by name
m = Measurement.get(session, "ASAM.M.SCALAR.UBYTE.IDENTICAL")
print(f"Name:       {m.name}")
print(f"Data type: {m.datatype}")
print(f"Address:   0x{m.ecuAddress:08X}")
print(f"Limits:    [{m.lowerLimit}, {m.upperLimit}]")
print(f"CompuMethod: {m.compuMethod.conversionType}")

# Convert a raw ECU value to physical
raw_value = 42
phys = m.compuMethod.int_to_physical(raw_value)
print(f"  {raw_value} raw → {phys} physical")

# Query all FLOAT32 measurements
float_meas = list(module.measurement.query(
    lambda row: row.datatype == "FLOAT32_IEEE"
))
print(f"\nFLOAT32 measurements: {len(float_meas)}")
for fm in float_meas:
    print(f"  {fm.name}")

# Query characteristics by type
maps = list(module.characteristic.query(
    lambda row: row.type == "MAP"
))
print(f"\nMAP characteristics: {len(maps)}")
for c in maps:
    print(f"  {c.name}: {c.num_axes} axes, shape={c.fnc_np_shape}")
```

2.6 Working with IF_DATA

IF_DATA sections carry vendor-specific protocol information (XCP, CCP, ...). The `if_data` attribute on every inspect object returns an `IfData` instance with parsed and raw data:

```
from pya2l import open_create
from pya2l.api.inspect import Module

session = open_create("xcp_demo_autodetect.a2l")
module = Module(session)

# Access the IfData dataclass
ifd = module.if_data

# Parsed structure (list of dicts)
for block in ifd.if_data_parsed:
```

(continues on next page)

(continued from previous page)

```
    print(block)

# Quick key look-up via flatmap
for key in ifd.flatmap:
    print(f"  {key}: {len(ifd.flatmap[key])} occurrence(s)")

# Raw text for debugging
for raw in ifd.if_data_raw:
    print(raw.raw[:200], "...")
```

See *Working with IF_DATA* for a comprehensive guide.

2.7 Validate and export

Basic validation:

```
from pya2l import open_existing
from pya2l.api.validate import Validator

session = open_existing("ASAP2_Demo_V161")
for msg in Validator(session)():
    print(msg.type.name, msg.category.name, msg.diag_code.name, "-", msg.text)
```

Export back to text:

```
from pya2l import export_a2l

export_a2l("ASAP2_Demo_V161", "exported.a2l")
```

See *HOW-TOs* for Excel export and other short recipes.

The test-suite found here is a good starting point for further experimentations, because it touches virtually every A2L element/attribute.

2.8 Next steps

- *Tutorial* — In-depth walkthrough of all features
- *Working with IF_DATA* — Comprehensive IF_DATA guide
- *API Reference* — Full API reference with examples
- *HOW-TOs* — Task-oriented quick recipes

2.9 Migrating from `pya2l.DB`

The `DB` wrapper class is **deprecated** and will be removed in a future release. Replace each call pattern as shown below — the behaviour is identical:

Old (deprecated)	New (preferred)
`DB().import_a2l("file.a2l")`	`import_a2l("file.a2l")`
`DB().open_existing("file")`	`open_existing("file")`
`DB().open_create("file.a2l")`	`open_create("file.a2l")`
`DB.import_a2l(db, "file.a2l")`	`import_a2l("file.a2l")`

All keyword arguments (`encoding`, `loglevel`, `local`, `progress_bar`, `force_overwrite`) are available on the module-level functions.

```
# Before
from pya2l import DB
session = DB().import_a2l("my.a2l", loglevel="ERROR")

# After
from pya2l import import_a2l
session = import_a2l("my.a2l", loglevel="ERROR")
```

CHAPTER

THREE

INSTALLATION

- Pythons: Python 3.10 (see pyproject classifiers for exact versions)
- Platforms: No platform-specific restrictions for installing the Python package; building from source may require additional tools (see Building).
- Documentation: Latest docs are in this repository under `docs/` as Markdown.

3.1 Prerequisites

If you plan to build from source, ensure you have a C/C++ toolchain and CMake installed (see Building). Java/ANTLR are not required for normal builds.

3.2 Install from PyPI

```
pip install pya2ldb
```

Important: The package name is `pya2ldb` (not `pya2l`).

3.3 Install from source (editable)

```
git clone https://github.com/christoph2/pyA2L.git
cd pyA2L
pip install -v -e .
```

CHAPTER

FOUR

BUILDING PYA2L (FROM SOURCE)

pyA2L ships binary wheels for common platforms on PyPI as `pya2ldb`. If a wheel is not available for your platform, you can build from source.

4.1 Prerequisites

- Python 3.10+
- A C/C++ toolchain (e.g., MSVC Build Tools on Windows, Xcode Command Line Tools on macOS, or GCC/Clang on Linux)
- CMake 3.12+
- pip >= 21.3

Note: The project uses pybind11 and CMake under the hood. ANTLR and Java are not required for normal builds.

4.2 One-liner build and install (recommended)

Using pip (PEP 517):

```
pip install -v .
```

This will compile the native extensions and install `pya2ldb` into your environment.

4.3 Development install

If you plan to work on the codebase, a development install keeps sources editable:

```
pip install -v -e .
```

This uses the build backend defined in `pyproject.toml` and will recompile extensions as needed.

4.4 Building distribution artifacts

Build a source distribution and wheel into the `dist/` directory:

```
python -m build
```

You can then upload with `twine`.

4.5 Building the documentation

The user guides live in `docs/` (mostly Markdown). The Sphinx entry point is `docs/index.rst`, which links to those pages for convenient browsing on GitHub.

To build the Sphinx site locally:

```
python -m pip install -r docs/requirements.txt sphinx
sphinx-build -b html docs docs/_build/html
```

Open `docs/_build/html/index.html` in your browser.

CHAPTER

FIVE

CONFIGURATION

pyA2L works out-of-the-box with sensible defaults. This section describes all available knobs for tuning import, export, and runtime behaviour.

5.1 Import options

`import_a2l()` (and `DB.import_a2l()`) accept these parameters:

Parameter	Default	Description
`file_name`	(required)	Path to the `.a2l` file.
`encoding`	`"latin-1"`	Character encoding of the A2L source. Common alternatives: `"utf-8"`, `"ascii"`, `"iso-8859-1"`. Use `chardetect file.a2l` on the command line to auto-detect.
`in_memory`	`False`	If `True`, the SQLite database is created in RAM (no `.a2ldb` file on disk). Faster, but the data is lost when the session closes.
`remove_existing`	`False`	If `True`, an existing `.a2ldb` file with the same name is deleted before importing.
`local`	`False`	If `True`, the `.a2ldb` is created in the current working directory instead of next to the source file.
`loglevel`	`"INFO"`	Python logging level. Set to `"ERROR"` or `"WARNING"` to suppress progress messages; `"DEBUG"` for verbose output.
`progress_bar`	`True`	Show a Rich progress bar during import. Disable with `False` for batch/CI usage.
`debug`	`False`	Enable parser-level debug output.

Example:

```
from pya2l import DB

session = DB().import_a2l(
    "my_ecu.a2l",
    encoding="utf-8",
    local=True,
    progress_bar=False,
    loglevel="WARNING",
)
```

5.2 Export options

`export_a2l()` parameters:

Parameter	Default	Description
`db_name`	(required)	Path to the `.a2ldb` database (extension implied if omitted).
`output`	`sys.stdout`	Output file path (string) or a writable file-like object.
`encoding`	`"latin1"`	Encoding for the output A2L text file.

Example:

```
from pya2l import export_a2l

# Export to file
export_a2l("my_ecu", "exported.a2l")

# Export to stdout
export_a2l("my_ecu")
```

5.3 Environment variables

Variable	Description
`ASAP_INCLUDE`	Platform-style separated list of directories used for resolving `/INCLUDE` file references in A2L files. Works like `PATH` on your operating system.

Example (Windows):

```
set ASAP_INCLUDE=C:\a2l\includes;D:\shared\a2l_libs
a2ldb-imex -i my_ecu.a2l
```

Example (Linux/macOS):

```
export ASAP_INCLUDE=/opt/a2l/includes:/shared/a2l_libs
a2ldb-imex -i my_ecu.a2l
```

5.4 CLI options (a2ldb-imex)

The `a2ldb-imex` console script provides quick command-line access:

```
a2ldb-imex -h            # show help
a2ldb-imex -V            # show version

# Import
a2ldb-imex -i file.a2l              # import (creates .a2ldb next to file)
a2ldb-imex -i file.a2l -L           # create .a2ldb in current directory
a2ldb-imex -i file.a2l -E utf-8     # specify encoding
a2ldb-imex -i file.a2l -p           # silence progress bar
```

(continues on next page)

(continued from previous page)

```
# Export
a2ldb-imex -e file.a2ldb -o out.a2l # export to file
a2ldb-imex -e file.a2ldb            # export to stdout

# JSON export
a2ldb-imex -e file.a2ldb --json -o out.json       # export as JSON
a2ldb-imex -e file.a2ldb --json --pretty -o out.json  # pretty-printed JSON
a2ldb-imex -e file.a2ldb --json                     # JSON to stdout
```

5.5 Database pragmas

pyA2L configures SQLite for optimal performance. These settings are applied automatically and normally do not need to be changed:

Pragma	Effect
`journal_mode=WAL`	Write-Ahead Logging — allows concurrent readers during writes
`FOREIGN_KEYS=ON`	Enforce foreign key constraints
`SYNCHRONOUS=OFF`	Faster writes (safe with WAL mode)
`busy_timeout=5000`	Wait up to 5 seconds when the database is locked

5.6 Logging

pyA2L uses Python's `logging` module. The main loggers are:

- `pya2l` — general library messages
- `pya2l.ifdata` — IF_DATA parsing diagnostics

Adjust verbosity:

```
import logging
logging.getLogger("pya2l").setLevel(logging.DEBUG)
logging.getLogger("pya2l.ifdata").setLevel(logging.WARNING)
```

CHAPTER

SIX

TUTORIAL

This tutorial provides a comprehensive guide to using the pyA2L library for working with ASAM A2L files. It covers basic usage, advanced features, and common use cases.

6.1 Basic Usage

6.1.1 Importing A2L Files

The first step in working with A2L files is to import them into a database. This allows for faster access and querying of the data.

```
from pya2l import DB

# Create a database instance
db = DB()

# Import an A2L file into a database
session = db.import_a2l("ASAP2_Demo_V161.a2l")
```

This creates a SQLite database file with the extension `.a2ldb` in your working directory.

6.1.2 Opening Existing Databases

If you've already imported an A2L file, you can open the existing database:

```
from pya2l import DB

db = DB()
session = db.open_existing("ASAP2_Demo_V161")  # No need to specify .a2ldb extension
```

Alternatively, you can use `open_create()` which will open an existing database if it exists, or create a new one if it doesn't:

```
session = db.open_create("ASAP2_Demo_V161.a2l")  # Creates database from A2L file
# or
session = db.open_create("ASAP2_Demo_V161")      # Opens existing database
```

6.1.3 Accessing Project Information

Once you have a session, you can access the project information:

```
from pya2l.api.inspect import Project

# Create a Project instance
project = Project(session)

# Access project attributes
print(project.name)
print(project.header.version)

# Access modules
for module in project.module:
    print(module.name)
```

6.2 Working with Modules

6.2.1 Accessing Module Elements

You can access various elements within a module:

```
# Get the first module
module = project.module[0]

# Access module attributes
print(module.name)
print(module.description)

# Access module elements using query methods
measurements = list(module.measurement.query())
characteristics = list(module.characteristic.query())
axis_points = list(module.axis_pts.query())
compu_methods = list(module.compu_method.query())
```

Understanding query()

All collections on Module (e.g., measurement, characteristic, axis_pts, compu_method, function, group, frame, unit, record_layout, ...) are FilteredList objects. Their .query() method: - Returns a generator of high-level inspect objects (e.g., Measurement, Characteristic), not raw ORM rows. Convert to a list if you need indexing or counting: list(...), len(list(...)). - Accepts an optional predicate function predicate(row) -> bool applied to the underlying ORM row (SQLAlchemy model), not to the inspect wrapper. Use fields as they appear in the A2L schema/DB (e.g., row.name, row.datatype, row.longIdentifier). - Iterates the association in Python and does not push filters down to SQL; for very large modules, consider issuing your own SQLAlchemy queries on pya2l.model.* if you need DB-side filtering.

Common patterns:

```
# 1) Get the first few measurements (materialize the generator)
meas_list = list(module.measurement.query())
first_three = meas_list[:3]

# 2) Find one by exact name (fast path using predicate on ORM rows)
name = "ENGINE_SPEED"
```

(continues on next page)

(continued from previous page)

```
found = next(module.measurement.query(lambda row: row.name == name), None)
if found:
    print("Found:", found.name, found.datatype)

# 3) Prefix or substring match
starts = list(module.characteristic.query(lambda row: row.name.startswith("ENGINE_")))
contains = list(module.characteristic.query(lambda row: "TEMP" in row.name))

# 4) Filter by datatype/limits
float_meas = list(module.measurement.query(
    lambda row: row.datatype in ("FLOAT32_IEEE", "FLOAT64_IEEE")
))

# 5) Count (remember query() is a generator)
count_meas = sum(1 for _ in module.measurement.query())

# 6) Sort client-side after materializing
sorted_meas = sorted(module.measurement.query(), key=lambda m: m.name)

# 7) Combine conditions
hi_res = list(module.measurement.query(
    lambda row: row.datatype == "UWORD" and (row.upperLimit or 0) > 1000
))
```

Notes and gotchas: - The predicate gets ORM rows. Attributes sometimes differ from the inspect object's property names. For example, Group rows use row.groupName and row.groupLongIdentifier; UserRights uses row.userLevelId. Module already wires these up (e.g., Module.group uses FilteredList(..., attr_name="groupName")) so you usually only care when writing predicates. - .query() yields inspect wrappers via Klass.get(session, key_attr). That means you can directly access high-level properties on results (e.g., m.physUnit, c.compuMethod, ax.record_layout). - For advanced filtering/ordering/pagination at the database level, query the ORM directly (session.query(pya2l.model.Measurement)...), then map names to inspect objects with Measurement.get(session, name) as needed.

6.2.2 Filtering Queries

You can filter queries using lambda functions:

```
# Get all measurements with FLOAT32_IEEE or FLOAT64_IEEE data types
float_measurements = list(module.measurement.query(
    lambda x: x.datatype in ("FLOAT32_IEEE", "FLOAT64_IEEE")
))

# Get all characteristics with a specific name pattern
specific_chars = list(module.characteristic.query(
    lambda x: x.name.startswith("ENGINE_")
))
```

6.3 Advanced Features

6.3.1 Working with IF_DATA Sections

IF_DATA sections contain vendor-specific information (XCP, CCP, KWP2000, ...). pyA2L parses these blocks and wraps the result in an `IfData` dataclass that gives you access to both the **parsed** and **raw** representation simultaneously.

The `IfData` dataclass

Every inspect object that can carry IF_DATA (`Module`, `Measurement`, `Characteristic`, `AxisPts`, `Function`, `Group`, `Frame`, `Instance`, `Blob`, `MemoryLayout`, `MemorySegment`) exposes an `if_data` attribute of type `IfData`:

- `if_data.if_data_parsed` — `List[Any]`: structured dicts produced by the AML-based parser, one entry per `/begin IF_DATA` block.
- `if_data.if_data_raw` — `list`: the original model objects whose `.raw` attribute holds the verbatim A2L text.
- `if_data.flatmap` — `Dict[str, List[Any]]`: lazily built flat index over all keys found in the parsed tree. Useful for quick look-ups when you know the tag but not the nesting depth.

Where you can find IF_DATA (inspect API):

- `Module.if_data`
- `Measurement.if_data`
- `Characteristic.if_data`
- `AxisPts.if_data`
- `Function.if_data`
- `Group.if_data`
- `Frame.if_data`
- `Instance.if_data`
- `Blob.if_data`
- `ModPar.memoryLayouts[i].if_data`
- `ModPar.memorySegments[i].if_data`

Accessing parsed IF_DATA

```
from pya2l.api.inspect import Module

module = Module(session)

# MODULE-level IF_DATA
print(module.if_data.if_data_parsed)  # list of parsed dicts

# MEASUREMENT IF_DATA
for meas in module.measurement.query():
    if meas.if_data.if_data_parsed:
        print(meas.name, meas.if_data.if_data_parsed)

# Quick tag look-up via flatmap
for meas in module.measurement.query():
    if "DAQ_LIST" in meas.if_data.flatmap:
```

(continues on next page)

(continued from previous page)

```
        print(f"{meas.name} has DAQ config: {meas.if_data.flatmap['DAQ_LIST']}")

# CHARACTERISTIC IF_DATA
for char in module.characteristic.query():
    if char.if_data.if_data_parsed:
        print(char.name, char.if_data.if_data_parsed)

# AXIS_PTS IF_DATA
for ax in module.axis_pts.query():
    if ax.if_data.if_data_parsed:
        print(ax.name, ax.if_data.if_data_parsed)

# FUNCTION / GROUP / FRAME IF_DATA
for fn in module.function.query():
    if fn.if_data.if_data_parsed:
        print(fn.name, fn.if_data.if_data_parsed)
for grp in module.group.query():
    if grp.if_data.if_data_parsed:
        print(grp.name, grp.if_data.if_data_parsed)
for fr in module.frame.query():
    if fr.if_data.if_data_parsed:
        print(fr.name, fr.if_data.if_data_parsed)

# MOD_PAR memory layouts/segments IF_DATA
mp = module.mod_par
if mp:
    for i, ml in enumerate(mp.memoryLayouts):
        if ml.if_data.if_data_parsed:
            print(f"MEMORY_LAYOUT[{i}]", ml.if_data.if_data_parsed)
    for i, ms in enumerate(mp.memorySegments):
        if ms.if_data.if_data_parsed:
            print(f"MEMORY_SEGMENT[{i}] {ms.name}", ms.if_data.if_data_parsed)
```

Accessing raw IF_DATA text

When the AML schema is unavailable or you need the verbatim text:

```
for meas in module.measurement.query():
    for raw_obj in meas.if_data.if_data_raw:
        print(f"--- {meas.name} raw IF_DATA ---")
        print(raw_obj.raw)
```

Parsing raw IF_DATA text manually

```
from pya2l.aml.ifdata_parser import IfDataParser

ifdata_parser = IfDataParser(session)

ifdata_text = """/begin IF_DATA XCP
/begin SEGMENT 0x01 0x02 0x00 0x00 0x00
/begin CHECKSUM XCP_ADD_44 MAX_BLOCK_SIZE 0xFFFF EXTERNAL_FUNCTION "" /end CHECKSUM
/begin PAGE 0x01 ECU_ACCESS_WITH_XCP_ONLY XCP_READ_ACCESS_WITH_ECU_ONLY XCP_WRITE_ACCESS_
```

(continues on next page)

(continued from previous page)

```
↪NOT_ALLOWED /end PAGE
/begin PAGE 0x00 ECU_ACCESS_WITH_XCP_ONLY XCP_READ_ACCESS_WITH_ECU_ONLY XCP_WRITE_ACCESS_
↪WITH_ECU_ONLY /end PAGE
/end SEGMENT
/end IF_DATA"""

parsed = ifdata_parser.parse(ifdata_text)
print(parsed)
```

Note

The inspect API always returns an `IfData` instance (never `None`). If no IF_DATA blocks exist, `if_data_parsed` and `if_data_raw` are empty lists. The `flatmap` property safely returns an empty dict.

For a comprehensive guide including XCP, CCP, and KWP2000 examples, see *Working with IF_DATA*.

6.3.2 Creating New A2L Elements

You can create new A2L elements using the creator classes:

```
from pya2l.api.create import CompuMethodCreator, MeasurementCreator

# Create a new computation method
cm_creator = CompuMethodCreator(session)
compu_method = cm_creator.create_compu_method(
    name="CM_LINEAR",
    long_identifier="Linear conversion",
    conversion_type="LINEAR",
    format_str="%.2f",
    unit="km/h"
)

# Add coefficients to the computation method
cm_creator.add_coeffs_linear(compu_method, a=0.1, b=0.0)

# Create a new measurement
meas_creator = MeasurementCreator(session)
measurement = meas_creator.create_measurement(
    name="ENGINE_SPEED",
    long_identifier="Engine speed",
    datatype="UWORD",
    compu_method="CM_LINEAR",
    lower_limit=0,
    upper_limit=8000,
    unit="rpm"
)

# Commit changes to the database
session.commit()
```

6.3.3 Coverage parity and additional creator examples

The Creator API (pya2l.api.create) aims for feature parity with the Inspector API (pya2l.api.inspect): every entity you can query should be possible to create. Below are examples for some commonly used creator methods recently added.

Create COMPU_TAB, COMPU_VTAB_RANGE, FRAME, TRANSFORMER, TYPEDEFs, and INSTANCE

```
from pya2l import DB
from pya2l.api.create import ModuleCreator
from pya2l.api.inspect import Module

session = DB().open_create("ASAP2_Demo_V161.a2l")

mc = ModuleCreator(session)
mod = mc.create_module("DEMO", "Demo ECU module")

# Numeric table conversion
ct = mc.add_compu_tab(
    mod, name="CT_DEMO", long_identifier="Demo numeric table",
    conversion_type="TAB_NOINTP",
    pairs=[(0, 0.0), (100, 1.0)],
    default_numeric=0.0,
)

# Verbal range conversion
vr = mc.add_compu_vtab_range(
    mod, name="VR_DEMO", long_identifier="State ranges",
    triples=[(0.0, 0.49, "OFF"), (0.5, 1.49, "ON"), (1.5, 10.0, "FAULT")],
    default_value="OFF",
)

# Frame with measurements
fr = mc.add_frame(
    mod, name="FRAME1", long_identifier="Example frame",
    scaling_unit=1, rate=10, measurements=["ENGINE_SPEED"],
)

# Transformer with in/out object lists
tr = mc.add_transformer(
    mod, name="TR1", version="1.0",
    dllname32="tr32.dll", dllname64="tr64.dll",
    timeout=1000, trigger="ON_CHANGE", reverse="NONE",
    in_objects=["ENGINE_SPEED"], out_objects=["SPEED_PHYS"],
)

# Typedef structure and a component
ts = mc.add_typedef_structure(mod, name="TSig", long_identifier="Signal", size=8)
mc.add_structure_component(ts, name="raw", type_ref="UWORD", offset=0)

# Instance of the typedef
inst = mc.add_instance(mod, name="S1", long_identifier="Inst of TSig",
                       type_name="TSig", address=0x1000)

# Inspect what we just created
```

(continues on next page)

(continued from previous page)

```
m = Module(session)
assert any(x.name == "CT_DEMO" for x in m.compu_tab.query())
assert any(x.name == "VR_DEMO" for x in m.compu_tab_verb_ranges.query())
assert any(x.name == "FRAME1" for x in m.frame.query())
assert any(x.name == "TR1" for x in m.transformer.query())
session.commit()
```

6.4 Working with Variant Coding

Variant coding allows for different configurations of the same ECU:

```
# Access variant coding information
variant_coding = module.variant_coding

# Print variant coding details
print(variant_coding.var_characteristic)
print(variant_coding.var_criterion)
print(variant_coding.var_forbidden_comb)
```

6.5 Common Use Cases

6.5.1 Extracting Measurement Information

A common task is to extract information about all measurements:

```
# Get all measurements
measurements = list(module.measurement.query())

# Print measurement details
for meas in measurements:
    print(f"Name: {meas.name}")
    print(f"Description: {meas.longIdentifier}")
    print(f"Data Type: {meas.datatype}")
    print(f"ECU Address: 0x{meas.address:08x}")
    print(f"Conversion: {meas.compuMethod.conversionType}")
    print(f"Unit: {meas.physUnit}")
    print("---")
```

6.5.2 Working with Characteristics

Characteristics represent calibration parameters:

```
# Get all characteristics
characteristics = list(module.characteristic.query())

# Print characteristic details
for char in characteristics:
    print(f"Name: {char.name}")
    print(f"Type: {char.type}")
    print(f"Address: 0x{char.address:08x}")
```

(continues on next page)

(continued from previous page)

```
    print(f"Record Layout: {char.depositAttr.name}")
    print("---")
```

6.5.3 Analyzing Record Layouts

Record layouts define how data is stored in memory:

```
# Get all record layouts
record_layouts = list(module.record_layout.query())

# Print record layout details
for rl in record_layouts:
    print(f"Name: {rl.name}")
    print(f"Alignment: {rl.alignment}")

    # Print components
    if rl.fnc_values:
        print(f"Function Values: {rl.fnc_values.position}, {rl.fnc_values.data_type}")

    if rl.axis_pts_x:
        print(f"X-Axis Points: {rl.axis_pts_x.position}, {rl.axis_pts_x.data_type}")

    if rl.axis_pts_y:
        print(f"Y-Axis Points: {rl.axis_pts_y.position}, {rl.axis_pts_y.data_type}")

    print("---")
```

6.6 Best Practices

1. Close Sessions: Always close your database sessions when you're done:

```
session.close()
```

2. Error Handling: Use try-except blocks to handle potential errors:

```
try:
    session = db.open_existing("NonExistentFile")
except Exception as e:
    print(f"Error opening database: {e}")
```

3. Commit Changes: When making changes to the database, remember to commit them:

```
# After making changes
session.commit()

# If something goes wrong, you can roll back
# session.rollback()
```

4. Use Query Filters: Filter your queries to improve performance:

```
# This is more efficient than getting all measurements and filtering in Python
float_measurements = list(module.measurement.query(
```

(continues on next page)

(continued from previous page)

```
    lambda x: x.datatype == "FLOAT32_IEEE"
))
```

5. Cache Results: For frequently accessed data, consider caching the results:

```
# Cache all measurements
all_measurements = list(module.measurement.query())

# Use the cached list instead of querying again
float_measurements = [m for m in all_measurements if m.datatype == "FLOAT32_IEEE"]
```

6.7 Conclusion

This tutorial covered the basics of working with pyA2L. For more detailed information, refer to the API reference documentation and the example scripts in the `pya2l/examples` directory.

CHAPTER

SEVEN

WORKING WITH IF_DATA

IF_DATA sections in A2L files carry vendor-specific transport-layer and protocol information (XCP, CCP, KWP2000, ...). pyA2L parses these blocks automatically when an AML description is available and wraps the result in the `IfData` dataclass, which gives you simultaneous access to the **parsed** (structured) and **raw** (text) representation.

7.1 The IfData class

Every inspect object that can contain IF_DATA blocks (`Module`, `Measurement`, `Characteristic`, `AxisPts`, `Function`, `Group`, `Frame`, `Instance`, `Blob`, and the `MemoryLayout` / `MemorySegment` entries inside `ModPar`) exposes an `if_data` attribute of type `IfData`.

```
from dataclasses import dataclass, field
from typing import Any, Dict, List, Union

@dataclass
class IfData:
    """Parsed + raw IF_DATA container.

    Parameters
    ----------
    if_data_parsed : List[Any]
        Structured representation produced by the AML-based parser.
        Each list element corresponds to one ``/begin IF_DATA ... /end IF_DATA``
        block in the source file.
    if_data_raw : list
        The original model objects (``pya2l.model.IfData``) that hold the
        raw text.  Useful when you need the verbatim A2L source, e.g. for
        re-export or manual inspection.

    Attributes
    ----------
    flatmap : Dict[str, List[Any]]
        A lazily built dictionary that flattens the nested parsed structure
        into a key → [values...] mapping.  Handy for quick look-ups when
        you know the tag name but not the nesting depth.
    """
    if_data_parsed: List[Any]
    if_data_raw: list
    items: Dict[str, List[Any]] = field(default_factory=dict)
```

(continues on next page)

(continued from previous page)

```
    @property
    def flatmap(self) -> Dict[str, List[Any]]: ...
```

7.2 Quick start

```
from pya2l import DB
from pya2l.api.inspect import Measurement

db = DB()
session = db.open_create("ASAP2_Demo_V161.a2l")

meas = Measurement.get(session, "ASAM.M.SCALAR.UBYTE.IDENTICAL")

# IfData instance - always present, may be empty
ifd = meas.if_data

# Parsed data (list of dicts/nested structures)
print(ifd.if_data_parsed)          # e.g. [{'XCP': { ... }}]

# Raw model objects (for re-export or text inspection)
print(len(ifd.if_data_raw))        # number of IF_DATA blocks

# Quick key-based look-up via flatmap
if "DAQ_LIST" in ifd.flatmap:
    print(ifd.flatmap["DAQ_LIST"])
```

7.3 Parsed data vs. raw data

Why both? The parsed representation is the fast path for programmatic access — you get Python dicts you can index into. The raw representation is essential when you need the original text for diagnostics, logging, or when the AML schema is unavailable (in which case `if_data_parsed` is empty but the raw text is still there).

7.4 Using the flatmap

Deeply nested IF_DATA trees can be tedious to traverse. `flatmap` flattens every key it encounters into a dictionary of lists:

```
from pya2l import DB
from pya2l.api.inspect import Module

session = DB().open_create("xcp_demo_autodetect.a2l")
module = Module(session)

ifd = module.if_data    # IfData instance

# Iterate all keys the parser found
for key, values in ifd.flatmap.items():
    print(f"{key}: {len(values)} occurrence(s)")
```

(continues on next page)

(continued from previous page)

```
# Direct look-up of a known tag
if "PROTOCOL_LAYER" in ifd.flatmap:
    proto = ifd.flatmap["PROTOCOL_LAYER"]
    print("Protocol layer info:", proto)

if "DAQ" in ifd.flatmap:
    daq = ifd.flatmap["DAQ"]
    print("DAQ configuration:", daq)
```

> **Note**
>
> `flatmap` is built lazily on first access. The cost is proportional to the depth of the parsed tree and is paid only once.

7.5 Accessing IF_DATA across all entity types

Every entity that can carry IF_DATA in the ASAP2 standard exposes it:

```
from pya2l import DB
from pya2l.api.inspect import (
    Module, Measurement, Characteristic, AxisPts,
    Function, Group, Frame,
)

session = DB().open_create("my_ecu.a2l")
module = Module(session)

# --- MODULE level ---
print("Module IF_DATA:", module.if_data.if_data_parsed)

# --- MEASUREMENT ---
for meas in module.measurement.query():
    if meas.if_data.if_data_parsed:
        print(f"  {meas.name}: {meas.if_data.if_data_parsed}")

# --- CHARACTERISTIC ---
for char in module.characteristic.query():
    if char.if_data.if_data_parsed:
        print(f"  {char.name}: {char.if_data.if_data_parsed}")

# --- AXIS_PTS ---
for ax in module.axis_pts.query():
    if ax.if_data.if_data_parsed:
        print(f"  {ax.name}: {ax.if_data.if_data_parsed}")

# --- FUNCTION, GROUP, FRAME ---
for fn in module.function.query():
    if fn.if_data.if_data_parsed:
        print(f"  FUNCTION {fn.name}: {fn.if_data.if_data_parsed}")

for grp in module.group.query():
```

(continues on next page)

(continued from previous page)

```
    if grp.if_data.if_data_parsed:
        print(f"  GROUP {grp.name}: {grp.if_data.if_data_parsed}")

for fr in module.frame.query():
    if fr.if_data.if_data_parsed:
        print(f"  FRAME {fr.name}: {fr.if_data.if_data_parsed}")

# --- MEMORY_LAYOUT / MEMORY_SEGMENT (inside ModPar) ---
mp = module.mod_par
if mp:
    for layout in mp.memoryLayouts:
        if layout.if_data.if_data_parsed:
            print(f"  MEMORY_LAYOUT: {layout.if_data.if_data_parsed}")
    for seg in mp.memorySegments:
        if seg.if_data.if_data_parsed:
            print(f"  MEMORY_SEGMENT {seg.name}: {seg.if_data.if_data_parsed}")
```

7.6 Practical examples

7.6.1 Extracting XCP transport-layer parameters

Many ECU projects use XCP on Ethernet or CAN. The transport configuration lives in the module-level IF_DATA:

```
from pya2l import DB
from pya2l.api.inspect import Module

session = DB().open_create("xcp_demo_autodetect.a2l")
module = Module(session)

ifd = module.if_data

# Walk the parsed tree for XCP specifics
for block in ifd.if_data_parsed:
    if not isinstance(block, dict):
        continue
    if "XCP" not in block:
        continue

    xcp = block["XCP"]

    # Protocol layer
    if "PROTOCOL_LAYER" in xcp:
        proto = xcp["PROTOCOL_LAYER"]
        print("XCP Protocol Layer:")
        print(f"  Version          : {proto.get('version', 'N/A')}")
        print(f"  T1 timeout [ms]  : {proto.get('T1', 'N/A')}")
        print(f"  Max CTO          : {proto.get('MAX_CTO', 'N/A')}")
        print(f"  Max DTO          : {proto.get('MAX_DTO', 'N/A')}")
        print(f"  Byte order       : {proto.get('BYTE_ORDER', 'N/A')}")

    # DAQ lists
```

(continues on next page)

(continued from previous page)

```
    if "DAQ" in xcp:
        daq = xcp["DAQ"]
        print(f"DAQ: {daq}")

    # Transport layer (TCP/UDP)
    for tl_key in ("TRANSPORT_LAYER_CMD", "XCP_ON_TCP_IP", "XCP_ON_UDP_IP", "XCP_ON_CAN
↪"):
        if tl_key in xcp:
            print(f"{tl_key}: {xcp[tl_key]}")
```

7.6.2 Inspecting CCP (CAN Calibration Protocol) blocks

CCP information typically appears under measurement or characteristic IF_DATA:

```
from pya2l import DB
from pya2l.api.inspect import Module

session = DB().open_create("If_ccp4.a2l")
module = Module(session)

# Collect all CCP-related IF_DATA across measurements
for meas in module.measurement.query():
    for block in meas.if_data.if_data_parsed:
        if isinstance(block, dict) and "ASAP1B_CCP" in block:
            ccp = block["ASAP1B_CCP"]
            print(f"{meas.name}: CCP info = {ccp}")

# Module-level CCP parameters
for block in module.if_data.if_data_parsed:
    if isinstance(block, dict) and "ASAP1B_CCP" in block:
        print("Module CCP:", block["ASAP1B_CCP"])
```

7.6.3 Working with KWP2000 transport data

KWP2000-based A2L files store baud rates, timing, and security parameters in IF_DATA:

```
from pya2l import DB
from pya2l.api.inspect import Module

session = DB().open_create("if_kwp6.a2l")
module = Module(session)

for block in module.if_data.if_data_parsed:
    if not isinstance(block, dict):
        continue
    if "ASAP1B_KWP2000" not in block:
        continue

    kwp = block["ASAP1B_KWP2000"]
    print("KWP2000 transport parameters:")

    # TP_BLOB contains baud rates, timing, SERAM, checksum config
```

(continues on next page)

(continued from previous page)

```
    if "TP_BLOB" in kwp:
        tp = kwp["TP_BLOB"]
        print(f"  TP_BLOB: {tp}")

    # SOURCE blocks describe measurement channels
    if "SOURCE" in kwp:
        sources = kwp["SOURCE"]
        if not isinstance(sources, list):
            sources = [sources]
        for src in sources:
            print(f"  SOURCE: {src}")
```

7.6.4 Accessing raw IF_DATA text

When the AML description is missing or you need the verbatim text:

```
from pya2l import DB
from pya2l.api.inspect import Measurement

session = DB().open_create("engine_ecu.a2l")
meas = Measurement.get(session, "EngineSpeed")

# Raw text of each IF_DATA block
for raw_obj in meas.if_data.if_data_raw:
    print("--- raw IF_DATA text ---")
    print(raw_obj.raw)
    print("-------------------------")
```

7.6.5 Combining parsed + raw for diagnostics

A typical diagnostic use-case: log both the parsed tree (for automated processing) and the original text (for human review):

```
import json
from pya2l import DB
from pya2l.api.inspect import Module

session = DB().open_create("my_project.a2l")
module = Module(session)

report = []
for meas in module.measurement.query():
    ifd = meas.if_data
    if not ifd.if_data_parsed:
        continue
    entry = {
        "name": meas.name,
        "parsed": ifd.if_data_parsed,
        "raw_texts": [obj.raw for obj in ifd.if_data_raw],
        "flatmap_keys": list(ifd.flatmap.keys()),
    }
    report.append(entry)
```

(continues on next page)

(continued from previous page)

```
# Write a diagnostic JSON report
with open("ifdata_report.json", "w") as f:
    json.dump(report, f, indent=2, default=str)

print(f"Wrote {len(report)} IF_DATA entries to ifdata_report.json")
```

7.7 Manual parsing with IfDataParser

If you have raw IF_DATA text from an external source (not from a parsed A2L), you can use `IfDataParser` directly. This requires a session whose AML schema has been loaded (i.e., the A2L file contained a `/begin A2ML ... /end A2ML` block):

```
from pya2l import DB
from pya2l.aml.ifdata_parser import IfDataParser

db = DB()
session = db.open_create("ASAP2_Demo_V161.a2l")

# Create the parser (reads AML from the session)
parser = IfDataParser(session)

# Parse a raw IF_DATA snippet
raw_text = """/begin IF_DATA XCP
    /begin SEGMENT 0x01 0x02 0x00 0x00 0x00
        /begin CHECKSUM XCP_ADD_44
            MAX_BLOCK_SIZE 0xFFFF
            EXTERNAL_FUNCTION ""
        /end CHECKSUM
        /begin PAGE 0x01
            ECU_ACCESS_WITH_XCP_ONLY
            XCP_READ_ACCESS_WITH_ECU_ONLY
            XCP_WRITE_ACCESS_NOT_ALLOWED
        /end PAGE
    /end SEGMENT
/end IF_DATA"""

result = parser.parse(raw_text)
print(result)
```

Note

`IfDataParser` requires a valid AML schema in the database. If no `/begin A2ML` block was present in the original A2L, `session.parse_ifdata()` returns an empty list and the IfData object will have `if_data_parsed == []`.

7.7.1 Using setup_ifdata_parser on the session

You can also set up IF_DATA parsing at the session level, which is what the inspect API does internally:

```python
from pya2l import DB

db = DB()
session = db.open_existing("ASAP2_Demo_V161")

# Initialize the IF_DATA parser for this session
session.setup_ifdata_parser(loglevel="DEBUG")

# Now session.parse_ifdata() is available
import pya2l.model as model
meas_obj = session.query(model.Measurement).filter_by(name="SomeMeasurement").first()
if meas_obj:
    parsed = session.parse_ifdata(meas_obj.if_data)
    print(parsed)
```

7.8 Empty IF_DATA and edge cases

Not all entities have IF_DATA, and not all A2L files contain AML schemas. The `IfData` object handles these gracefully:

```python
from pya2l import DB
from pya2l.api.inspect import Measurement

session = DB().open_create("simple_file.a2l")
meas = Measurement.get(session, "SomeSignal")

ifd = meas.if_data

# No IF_DATA blocks → empty lists
if not ifd.if_data_parsed:
    print("No parsed IF_DATA available")

if not ifd.if_data_raw:
    print("No raw IF_DATA blocks")

# flatmap is an empty dict
assert ifd.flatmap == {} or len(ifd.flatmap) == 0

# Safe to iterate - no exceptions
for key, vals in ifd.flatmap.items():
    print(key, vals)
```

7.9 Best practices

1. **Check ``if_data_parsed`` first**: It's the convenient, structured representation. Fall back to `if_data_raw` only when you need the original text or the AML-based parsing is unavailable.
2. **Use ``flatmap`` for tag-based look-ups**: Instead of walking nested dicts manually, use `flatmap["TAG_NAME"]` when you know the key.
3. **Iterate over ``if_data_raw`` for debugging**: The `.raw` attribute gives you the exact text from the A2L file — invaluable when the parsed structure looks unexpected.

4. **Guard for empty data**: Always check `if ifd.if_data_parsed:` before accessing elements. Empty IfData objects are normal and frequent.
5. **AML is required for parsing**: Without an AML schema in the A2L file, `if_data_parsed` will be empty even if raw IF_DATA blocks exist. Use `if_data_raw` in that case.

CHAPTER

EIGHT

API REFERENCE

This reference covers the three public API modules:

- **Inspect** (`pya2l.api.inspect`) — read-only access to A2L entities
- **Create** (`pya2l.api.create`) — build or augment A2L databases
- **Validate** (`pya2l.api.validate`) — run diagnostic checks

All APIs operate on a `SessionProxy` obtained via `pya2l.import_a2l()`, `pya2l.open_existing()`, or `pya2l.open_create()`.

On this page

- *Session Management*
- *Inspect API*
  - *Project and Module*
  - *FilteredList and query()*
  - *CachedBase and .get()*
  - *Measurement*
  - *Characteristic*
  - *AxisPts*
  - *CompuMethod*
  - *RecordLayout*
  - *Function and Group*
  - *ModCommon and ModPar*
  - *Frame*
  - *VariantCoding*
  - *IfData*
- *Create API*
  - *ProjectCreator*
  - *ModuleCreator*
  - *CompuMethodCreator*

        - *MeasurementCreator*
        - *CharacteristicCreator*
        - *RecordLayoutCreator*
        - *FunctionCreator*
        - *GroupCreator*
        - *Complete example: building an ECU database from scratch*
    - *Validate API*
        - *Message fields*
        - *Diagnostic codes*
    - *Low-Level ORM Access*

8.1 Session Management

Every workflow starts by obtaining a session:

```
from pya2l import DB

db = DB()

# Parse A2L → SQLite  (creates .a2ldb)
session = db.import_a2l("ASAP2_Demo_V161.a2l")

# Open existing database  (fast, no reparsing)
session = db.open_existing("ASAP2_Demo_V161")

# Open-or-create  (convenience method)
session = db.open_create("ASAP2_Demo_V161.a2l")
```

The returned `session` is a `SessionProxy` that wraps a SQLAlchemy `Session` and adds IF_DATA parsing capabilities. You can use it for raw SQLAlchemy queries *and* for the high-level inspect/create/validate APIs.

Function-level API (no `DB` class needed):

```
from pya2l import import_a2l, open_existing, export_a2l

session = import_a2l("file.a2l", encoding="utf-8", progress_bar=False)
session = open_existing("file")
export_a2l("file", "output.a2l")
```

8.2 Inspect API

The inspect API lives in `pya2l.api.inspect` and provides read-only, high-level wrappers around the ORM model.

8.2.1 Project and Module

`Project` is the top-level entry point. It contains one or more `Module` objects, each holding all A2L entities.

```
from pya2l.api.inspect import Project, Module

project = Project(session)
print(project.name)            # e.g. "DemoProject"
print(project.header.version)  # e.g. "1.0"

# Iterate modules
for mod in project.module:
    print(mod.name, mod.longIdentifier)

# Direct access if you know the module name
module = Module(session, "MyModule")
```

`Module` attributes (all are `FilteredList` unless noted):

Attribute	Type / Description
axis_pts	FilteredList[AxisPts]
blob	FilteredList[Blob]
characteristic	FilteredList[Characteristic]
compu_method	FilteredList[CompuMethod]
compu_tab	FilteredList[CompuTab]
compu_tab_verb	FilteredList[CompuTabVerb]
compu_tab_verb_ranges	FilteredList[CompuTabVerbRanges]
frame	FilteredList[Frame]
function	FilteredList[Function]
group	FilteredList[Group]
measurement	FilteredList[Measurement]
record_layout	FilteredList[RecordLayout]
transformer	FilteredList[Transformer]
unit	FilteredList[Unit]
mod_common	ModCommon — byte order, alignment, data size
mod_par	ModPar — ECU metadata, memory segments, system constants
variant_coding	VariantCoding or None
if_data	IfData — module-level IF_DATA

8.2.2 FilteredList and query()

All collections on `Module` are `FilteredList` objects. Use `.query()` to iterate high-level inspect wrappers:

```
# All measurements (generator → list)
all_meas = list(module.measurement.query())

# With predicate (applied to ORM row, not inspect object)
float_meas = list(module.measurement.query(
    lambda row: row.datatype in ("FLOAT32_IEEE", "FLOAT64_IEEE")
))

# Prefix match
engine = list(module.characteristic.query(
```

(continues on next page)

(continued from previous page)

```
    lambda row: row.name.startswith("ENGINE_")
))

# Count without materialising
n = sum(1 for _ in module.measurement.query())

# Sort after materialising
by_name = sorted(module.measurement.query(), key=lambda m: m.name)
```

Important

The predicate receives the **ORM row** (`pya2l.model.*`), not the inspect wrapper. Use the schema field names (e.g. `row.name`, `row.datatype`, `row.groupName`).

8.2.3 CachedBase and .get()

All entity classes inherit from `CachedBase` which maintains a per-session LRU cache. **Always use `.get()`** instead of the constructor:

```
from pya2l.api.inspect import Measurement, Characteristic, AxisPts

meas = Measurement.get(session, "ENGINE_SPEED")
char = Characteristic.get(session, "InjectionMap")
axis = AxisPts.get(session, "RPM_Axis")

# Cache is transparent - second call returns same object
assert Measurement.get(session, "ENGINE_SPEED") is meas
```

8.2.4 Measurement

Read-only access to MEASUREMENT entities.

```
from pya2l.api.inspect import Measurement

m = Measurement.get(session, "ENGINE_SPEED")

# Core attributes
print(m.name)               # "ENGINE_SPEED"
print(m.longIdentifier)     # "Engine rotational speed"
print(m.datatype)           # "UWORD"
print(m.resolution)         # 1
print(m.accuracy)           # 0.5
print(m.lowerLimit)         # 0.0
print(m.upperLimit)         # 8000.0

# Address
print(f"0x{m.ecuAddress:08X}")     # e.g. 0x00100000
print(m.ecuAddressExtension)       # 0

# Conversion (CompuMethod)
print(m.compuMethod.name)          # "CM_Speed"
```

(continues on next page)

(continued from previous page)

```
print(m.compuMethod.conversionType)  # "LINEAR"
raw_value = 1200
phys = m.compuMethod.int_to_physical(raw_value)
print(f"{raw_value} → {phys} {m.physUnit}")  # "1200 → 300.0 rpm"

# Optional attributes
print(m.byteOrder)          # "LITTLE_ENDIAN" or None
print(m.bitMask)            # e.g. 0xFFFF or None
print(m.format)             # e.g. "%8.2" or None
print(m.displayIdentifier)  # e.g. "EngSpd" or None
print(m.symbolLink)         # SymbolLink namedtuple or None
print(m.arraySize)          # int or None
print(m.matrixDim)          # MatrixDim dataclass
print(m.discrete)           # bool
print(m.readWrite)          # bool
print(m.annotations)        # list of Annotation
print(m.functionList)       # list of function names
print(m.virtual)            # list of measuring channels
print(m.if_data)            # IfData instance

# NumPy shape (for arrays/matrices)
print(m.fnc_np_shape)       # e.g. (8, 4)
```

8.2.5 Characteristic

Read-only access to CHARACTERISTIC entities (VALUE, CURVE, MAP, CUBOID, ...).

```
from pya2l.api.inspect import Characteristic

c = Characteristic.get(session, "InjectionMap")

# Core
print(c.name)            # "InjectionMap"
print(c.type)            # "MAP"
print(f"0x{c.address:08X}")
print(c.maxDiff)
print(c.lowerLimit, c.upperLimit)

# Conversion
print(c.compuMethod.name)
print(c.compuMethod.int_to_physical(42))

# Record layout
rl = c.deposit           # RecordLayout instance
print(rl.name)
print(rl.fncValues)      # FncValues(position=..., data_type=..., ...)
print(rl.axes)           # dict of axis components

# Axis descriptions
for i, ax in enumerate(c.axisDescriptions):
    print(f"Axis {i}: {ax.attribute}, max={ax.maxAxisPoints}")
    print(f"  CompuMethod: {ax.compuMethod.name}")
```

(continues on next page)

(continued from previous page)

```
    print(f"  Limits: [{ax.lowerLimit}, {ax.upperLimit}]")

# Number of axes
print(c.num_axes)        # 2 for MAP

# Shape
print(c.fnc_np_shape)    # e.g. (16, 16)
print(c.dim)             # total number of function values

# Memory layout
print(c.record_layout_components)
print(f"Total memory: {c.total_allocated_memory} bytes")

# Optional attributes
print(c.calibrationAccess)  # e.g. "CALIBRATION"
print(c.encoding)           # e.g. "UTF8" or None
print(c.matrixDim)          # MatrixDim
print(c.extendedLimits)     # ExtendedLimits or None
print(c.if_data)            # IfData instance

# Dependent / virtual characteristics
print(c.dependent_characteristic)
print(c.virtual_characteristic)
```

8.2.6 AxisPts

Shared axis definitions used by multiple characteristics.

```
from pya2l.api.inspect import AxisPts

ax = AxisPts.get(session, "RPM_Axis")

print(ax.name)              # "RPM_Axis"
print(ax.longIdentifier)
print(f"0x{ax.address:08X}")
print(ax.maxAxisPoints)     # e.g. 16
print(ax.lowerLimit, ax.upperLimit)

# Conversion
print(ax.compuMethod.name)

# Record layout
print(ax.depositAttr.name)  # RecordLayout name
print(ax.record_layout_components)

# Memory
print(f"Allocated: {ax.total_allocated_memory} bytes")

# IF_DATA
print(ax.if_data)           # IfData instance
```

8.2.7 CompuMethod

Conversion methods translate internal (raw) ECU values to physical values and back.

```
from pya2l.api.inspect import CompuMethod

cm = CompuMethod.get(session, "CM_Speed")

print(cm.name)              # "CM_Speed"
print(cm.conversionType)    # "LINEAR", "RAT_FUNC", "TAB_VERB", ...
print(cm.format)            # "%8.2"
print(cm.unit)              # "rpm"

# Forward / inverse conversion
phys = cm.int_to_physical(1200)
raw  = cm.physical_to_int(300.0)

# Type-specific attributes
if cm.conversionType == "LINEAR":
    print(cm.coeffs_linear)  # CoeffsLinear(a=..., b=...)
elif cm.conversionType == "RAT_FUNC":
    print(cm.coeffs)         # Coeffs(a, b, c, d, e, f)
elif cm.conversionType == "FORM":
    print(cm.formula)        # {'formula': '...', 'formula_inv': '...'}
elif cm.conversionType in ("TAB_INTP", "TAB_NOINTP"):
    print(cm.tab)            # CompuTab
elif cm.conversionType == "TAB_VERB":
    print(cm.tab_verb)       # CompuTabVerb
```

Supported types:

Conversion Type	Description
IDENTICAL	Pass-through (phys = raw)
LINEAR	phys = a·x + b
RAT_FUNC	phys = (a·x² + b·x + c) / (d·x² + e·x + f)
FORM	Arbitrary formula string (evaluated via numexpr)
TAB_INTP	Table with interpolation
TAB_NOINTP	Table without interpolation (nearest match)
TAB_VERB	Verbal table (numeric → text)
NO_COMPU_METHOD	Explicitly no conversion

8.2.8 RecordLayout

Describes the physical memory structure of characteristics and axis points.

```
from pya2l.api.inspect import RecordLayout

rl = RecordLayout.get(session, "RL_MAP_16x16")

print(rl.name)
print(rl.alignment)              # Alignment dataclass

# Function values
```

(continues on next page)

(continued from previous page)

```
fv = rl.fncValues
if fv.valid():
    print(f"FNC_VALUES: pos={fv.position}, type={fv.data_type}")
    print(f"  NumPy dtype : {rl.fnc_np_dtype}")
    print(f"  Element size: {rl.fnc_element_size} bytes")

# Axes (dict: axis_name → dict of components)
for axis_name, components in rl.axes.items():
    print(f"Axis '{axis_name}':")
    for comp_name, comp in components.items():
        print(f"  {comp_name}: {comp}")

# Identification
if rl.identification.valid():
    print(f"ID: pos={rl.identification.position}, type={rl.identification.data_type}")
```

8.2.9 Function and Group

`Function` represents logical groupings of calibration parameters. `Group` defines UI folder structures.

```
from pya2l.api.inspect import Function, Group

# --- Functions ---
fn = Function.get(session, "InjectionControl")
print(fn.name)
print(fn.longIdentifier)
print(fn.annotations)
print(fn.functionVersion)

# Associated elements
print(fn.defCharacteristics)    # list of Characteristic
print(fn.inMeasurements)        # list of Measurement
print(fn.outMeasurements)       # list of Measurement
print(fn.locMeasurements)       # list of Measurement
print(fn.subFunctions)          # list of Function

# Get all top-level functions
roots = Function.get_root_functions(session, ordered=True)

# --- Groups ---
grp = Group.get(session, "EngineParams")
print(grp.name)
print(grp.root)                 # True if top-level group
print(grp.characteristics)      # list of Characteristic/AxisPts
print(grp.measurements)         # list of Measurement
print(grp.functions)            # list of Function
print(grp.subgroups)            # list of Group

# Get all root groups
roots = Group.get_root_groups(session, ordered=True)
```

8.2.10 ModCommon and ModPar

Module-wide settings and ECU metadata.

```
from pya2l.api.inspect import Module

module = Module(session)

# --- ModCommon ---
mc = module.mod_common
print(mc.comment)
print(mc.byteOrder)        # "LITTLE_ENDIAN" or "BIG_ENDIAN"
print(mc.dataSize)         # default word width
print(mc.alignment)        # Alignment dataclass
print(mc.alignment.byte)   # e.g. 1
print(mc.alignment.word)   # e.g. 2
print(mc.alignment.dword)  # e.g. 4

# --- ModPar ---
mp = module.mod_par
if mp:
    print(mp.comment)
    print(mp.cpu)           # e.g. "TC1766"
    print(mp.customer)
    print(mp.ecu)
    print(mp.epk)           # list of EPK strings
    print(mp.version)

    # System constants
    for name, value in mp.systemConstants.items():
        print(f"  {name} = {value}")

    # Memory segments
    for seg in mp.memorySegments:
        print(f"  {seg.name}: {seg.memoryType} @ 0x{seg.address:08X}, "
              f"size={seg.size}")
```

8.2.11 Frame

```
from pya2l.api.inspect import Frame

fr = Frame.get(session, "FRAME1")
print(fr.name)
print(fr.longIdentifier)
print(fr.scalingUnit)
print(fr.rate)
print(fr.frame_measurement)   # list of measurement names
print(fr.if_data)             # IfData instance
```

8.2.12 VariantCoding

Access variant definitions for ECUs with multiple calibration data-sets:

```
module = Module(session)
vc = module.variant_coding

if vc and vc.variant_coded:
    print(f"Separator: {vc.separator}")
    print(f"Naming   : {vc.naming}")

    # Available criterion values
    for crit_name in vc.criterions:
        values = vc.get_citerion_values(crit_name)
        print(f"  {crit_name}: {values}")

    # All valid combinations
    combos = vc.valid_combinations(list(vc.criterions.keys()))
    for combo in combos:
        print(combo)

    # Variants of a specific characteristic
    variants = vc.variants("MyCalibrationParam")
    for v in variants:
        print(v)
```

8.2.13 IfData

See *Working with IF_DATA* for a full guide. Quick reference:

```
from pya2l.api.inspect import Measurement

m = Measurement.get(session, "ENGINE_SPEED")

ifd = m.if_data                    # IfData instance
ifd.if_data_parsed                 # List[Any] - structured dicts
ifd.if_data_raw                    # list - original model objects
ifd.flatmap                        # Dict[str, List[Any]] - lazy flat index
```

8.3 Create API

The create API lives in `pya2l.api.create` and provides builder classes for constructing A2L databases programmatically.

All creators follow the same pattern:

1. Instantiate with a session
2. Call `create_*` / `add_*` methods
3. Call `commit()`

8.3.1 ProjectCreator

```
from pya2l.api.create import ProjectCreator

pc = ProjectCreator(session)
project = pc.create_project("DemoProject", "Example ECU calibration")
header = pc.add_header(project, "File comment / description")
pc.add_project_no(header, "PRJ-001")
pc.commit()
```

8.3.2 ModuleCreator

Creates modules and most structural elements (units, groups, functions, frames, transformers, typedefs, instances, ...):

```
from pya2l.api.create import ModuleCreator

mc = ModuleCreator(session)
module = mc.create_module("Engine", "Engine control", project=project)

# Common settings
mc.add_mod_common(module, "Module comment",
    byte_order="LITTLE_ENDIAN", data_size=32)

# Units
mc.add_unit(module, "rpm", "Revolutions per minute",
            display="rpm", type_str="DERIVED")
mc.add_unit(module, "degC", "Degrees Celsius",
            display="°C", type_str="DERIVED")

# Groups
grp = mc.add_group(module, "EngineParams", "Engine parameters")
mc.add_group_ref_measurement(grp, ["EngineSpeed", "CoolantTemp"])
mc.add_group_ref_characteristic(grp, ["IdleSpeed"])

# Frames
mc.add_frame(module, "Frame1", "CAN frame",
             scaling_unit=1, rate=10,
             measurements=["EngineSpeed"])

# Typedef structures
ts = mc.add_typedef_structure(module, "TSig", "Signal struct", size=4)
mc.add_structure_component(ts, "raw", "UWORD", offset=0)
mc.add_structure_component(ts, "status", "UBYTE", offset=2)

# Instances
mc.add_instance(module, "Signal1", "Instance of TSig",
                type_name="TSig", address=0x1000)

mc.commit()
```

8.3.3 CompuMethodCreator

```
from pya2l.api.create import CompuMethodCreator

cmc = CompuMethodCreator(session)

# LINEAR: phys = factor * raw + offset
cm = cmc.create_compu_method(
    "CM_Temp", "Temperature", "LINEAR",
    "%6.1", "°C", module_name="Engine"
)
cmc.add_coeffs_linear(cm, offset=-40.0, factor=0.1)

# RAT_FUNC: rational polynomial
cm2 = cmc.create_compu_method(
    "CM_Pressure", "Pressure", "RAT_FUNC",
    "%8.3", "bar", module_name="Engine"
)
cmc.add_coeffs(cm2, a=0, b=0.01, c=-1.0, d=0, e=0, f=1)

# TAB_VERB: enumeration
cm3 = cmc.create_compu_method(
    "CM_GearPos", "Gear", "TAB_VERB",
    "%d", "", module_name="Engine"
)

# Numeric table
ct = mc.add_compu_tab(
    module, "CT_Correction", "Correction factors",
    conversion_type="TAB_NOINTP",
    pairs=[(0, 1.0), (50, 1.05), (100, 1.12)],
    default_numeric=1.0,
)

# Verbal range table
vr = mc.add_compu_vtab_range(
    module, "VR_State", "Operating states",
    triples=[(0.0, 0.9, "OFF"), (1.0, 1.9, "STANDBY"), (2.0, 10.0, "RUN")],
    default_value="UNKNOWN",
)

cmc.commit()
```

8.3.4 MeasurementCreator

```
from pya2l.api.create import MeasurementCreator

mec = MeasurementCreator(session)

# Basic scalar measurement
m = mec.create_measurement(
    "EngineSpeed", "Engine RPM",
    "UWORD", "CM_Speed",
```

(continues on next page)

(continued from previous page)

```
    resolution=1, accuracy=1.0,
    lower_limit=0.0, upper_limit=8000.0,
    module_name="Engine"
)
mec.add_ecu_address(m, 0x100000)
mec.add_format(m, "%8.2")
mec.add_byte_order(m, "LITTLE_ENDIAN")
mec.add_display_identifier(m, "EngSpd")

# Matrix measurement
m2 = mec.create_measurement(
    "TempGrid", "Temperature sensor array",
    "SWORD", "CM_Temp",
    resolution=1, accuracy=0.1,
    lower_limit=-40.0, upper_limit=150.0,
    module_name="Engine"
)
mec.add_matrix_dim(m2, dims=[4, 4])
mec.add_ecu_address(m2, 0x200000)

# Symbol-linked measurement (no direct address)
m3 = mec.create_measurement(
    "VirtualSignal", "Computed signal",
    "FLOAT32_IEEE", "CM_Voltage",
    resolution=1, accuracy=0.01,
    lower_limit=0.0, upper_limit=5.0,
    module_name="Engine"
)
mec.add_symbol_link(m3, "g_voltage_sensor", offset=0)

mec.commit()
```

8.3.5 CharacteristicCreator

```
from pya2l.api.create import CharacteristicCreator

cc = CharacteristicCreator(session)

# VALUE (scalar)
c1 = cc.create_characteristic(
    "IdleSpeed", "Target idle RPM",
    "VALUE", 0x300000, "RL_UWORD", 0.0, "CM_Speed",
    500.0, 1500.0, module_name="Engine"
)

# CURVE (1-D)
c2 = cc.create_characteristic(
    "ThrottleCurve", "Throttle map",
    "CURVE", 0x310000, "RL_CURVE", 0.0, "CM_Percent",
    0.0, 100.0, module_name="Engine"
)
```

(continues on next page)

(continued from previous page)

```
cc.add_axis_descr(c2, "STD_AXIS", "NO_INPUT_QUANTITY",
                  "CM_Percent", 8, 0.0, 100.0)

# MAP (2-D)
c3 = cc.create_characteristic(
    "InjectionMap", "Fuel injection",
    "MAP", 0x320000, "RL_MAP", 0.0, "CM_Time",
    0.0, 50.0, module_name="Engine"
)
cc.add_axis_descr(c3, "STD_AXIS", "EngineSpeed",
                  "CM_Speed", 16, 0.0, 8000.0)
cc.add_axis_descr(c3, "STD_AXIS", "EngineLoad",
                  "CM_Percent", 16, 0.0, 100.0)

cc.commit()
```

8.3.6 RecordLayoutCreator

```
from pya2l.api.create import RecordLayoutCreator

rlc = RecordLayoutCreator(session)

# Scalar layout
rl = rlc.create_record_layout("RL_UWORD", module_name="Engine")
rlc.add_fnc_values(rl, position=1, datatype="UWORD",
                   index_mode="ALTERNATE", address_type="DIRECT")

# Curve layout (axis + values)
rl2 = rlc.create_record_layout("RL_CURVE", module_name="Engine")
rlc.add_no_axis_pts_x(rl2, position=1, datatype="UBYTE")
rlc.add_axis_pts_x(rl2, position=2, datatype="UWORD",
                   index_incr="INDEX_INCR", address_type="DIRECT")
rlc.add_fnc_values(rl2, position=3, datatype="UWORD",
                   index_mode="ALTERNATE", address_type="DIRECT")

# Map layout (2 axes + values)
rl3 = rlc.create_record_layout("RL_MAP", module_name="Engine")
rlc.add_no_axis_pts_x(rl3, position=1, datatype="UBYTE")
rlc.add_no_axis_pts_y(rl3, position=2, datatype="UBYTE")
rlc.add_axis_pts_x(rl3, position=3, datatype="UWORD",
                   index_incr="INDEX_INCR", address_type="DIRECT")
rlc.add_axis_pts_y(rl3, position=4, datatype="UWORD",
                   index_incr="INDEX_INCR", address_type="DIRECT")
rlc.add_fnc_values(rl3, position=5, datatype="UWORD",
                   index_mode="ALTERNATE", address_type="DIRECT")

# Alignment settings
rlc.add_alignment_byte(rl3, 1)
rlc.add_alignment_word(rl3, 2)
rlc.add_alignment_long(rl3, 4)

rlc.commit()
```

8.3.7 FunctionCreator

```
from pya2l.api.create import FunctionCreator

fc = FunctionCreator(session)

fn = fc.create_function(
    "InjectionCtrl", "Injection control",
    module_name="Engine"
)
fc.add_function_version(fn, "1.2.0")
fc.add_def_characteristic(fn, ["InjectionMap", "IdleSpeed"])
fc.add_in_measurement(fn, ["EngineSpeed", "EngineLoad"])
fc.add_out_measurement(fn, ["InjectorDuty"])
fc.add_sub_function(fn, ["ColdStartEnrich"])

fc.commit()
```

8.3.8 GroupCreator

```
from pya2l.api.create import GroupCreator

gc = GroupCreator(session)

grp = gc.create_group("EngineBasic", "Basic engine parameters",
                      module_name="Engine")
gc.add_root(grp)
gc.add_ref_characteristic(grp, ["IdleSpeed", "ThrottleCurve"])
gc.add_ref_measurement(grp, ["EngineSpeed"])
gc.add_sub_group(grp, ["AdvancedParams"])

gc.commit()
```

8.3.9 Complete example: building an ECU database from scratch

```
from pya2l import DB, export_a2l
from pya2l.api.create import (
    ProjectCreator, ModuleCreator, CompuMethodCreator,
    MeasurementCreator, CharacteristicCreator,
    RecordLayoutCreator, FunctionCreator, GroupCreator,
)
from pya2l.api.inspect import Project

# --- Setup ---
db = DB()
session = db.open_create("TurboECU.a2ldb")

pc = ProjectCreator(session)
project = pc.create_project("TurboECU", "Turbocharged engine ECU")
hdr = pc.add_header(project, "Created by pyA2L automation")
pc.add_project_no(hdr, "TE-2026-001")
```

(continues on next page)

(continued from previous page)

```
mc = ModuleCreator(session)
module = mc.create_module("TCU", "Turbo control unit", project=project)
mc.add_mod_common(module, "Common settings",
                  byte_order="LITTLE_ENDIAN", data_size=32)

# --- Units ---
mc.add_unit(module, "rpm",  "Revolutions per minute", "rpm", "DERIVED")
mc.add_unit(module, "bar",  "Pressure",               "bar", "DERIVED")
mc.add_unit(module, "degC", "Temperature",             "°C",  "DERIVED")

# --- Conversions ---
cmc = CompuMethodCreator(session)
cm_rpm = cmc.create_compu_method(
    "CM_RPM", "RPM", "LINEAR", "%8.0", "rpm", module_name="TCU")
cmc.add_coeffs_linear(cm_rpm, offset=0.0, factor=1.0)

cm_bar = cmc.create_compu_method(
    "CM_Boost", "Boost pressure", "LINEAR", "%5.2", "bar", module_name="TCU")
cmc.add_coeffs_linear(cm_bar, offset=-1.0, factor=0.01)

cm_temp = cmc.create_compu_method(
    "CM_Temp", "Temperature", "LINEAR", "%6.1", "°C", module_name="TCU")
cmc.add_coeffs_linear(cm_temp, offset=-40.0, factor=0.1)

# --- Record Layouts ---
rlc = RecordLayoutCreator(session)
rl_u16 = rlc.create_record_layout("RL_U16", module_name="TCU")
rlc.add_fnc_values(rl_u16, 1, "UWORD", "ALTERNATE", "DIRECT")

rl_curve = rlc.create_record_layout("RL_CURVE_12", module_name="TCU")
rlc.add_no_axis_pts_x(rl_curve, 1, "UBYTE")
rlc.add_axis_pts_x(rl_curve, 2, "UWORD", "INDEX_INCR", "DIRECT")
rlc.add_fnc_values(rl_curve, 3, "UWORD", "ALTERNATE", "DIRECT")

# --- Measurements ---
mec = MeasurementCreator(session)

m_rpm = mec.create_measurement(
    "EngineSpeed", "Current engine speed",
    "UWORD", "CM_RPM", 1, 1.0, 0.0, 8000.0, module_name="TCU")
mec.add_ecu_address(m_rpm, 0x100000)

m_boost = mec.create_measurement(
    "BoostPressure", "Turbo boost pressure",
    "UWORD", "CM_Boost", 1, 0.1, -1.0, 3.5, module_name="TCU")
mec.add_ecu_address(m_boost, 0x100002)

m_temp = mec.create_measurement(
    "ChargeAirTemp", "Charge air temperature",
    "UWORD", "CM_Temp", 1, 0.5, -40.0, 150.0, module_name="TCU")
mec.add_ecu_address(m_temp, 0x100004)
```

(continues on next page)

(continued from previous page)

```
# --- Characteristics ---
cc = CharacteristicCreator(session)

c_target = cc.create_characteristic(
    "TargetBoost", "Target boost pressure",
    "VALUE", 0x200000, "RL_U16", 0.0, "CM_Boost",
    0.0, 3.0, module_name="TCU")

c_curve = cc.create_characteristic(
    "BoostCurve", "Boost vs RPM",
    "CURVE", 0x201000, "RL_CURVE_12", 0.0, "CM_Boost",
    0.0, 3.0, module_name="TCU")
cc.add_axis_descr(c_curve, "STD_AXIS", "EngineSpeed",
                  "CM_RPM", 12, 0.0, 8000.0)

# --- Organisation ---
fc = FunctionCreator(session)
fn = fc.create_function("BoostControl", "Boost control logic",
                        module_name="TCU")
fc.add_def_characteristic(fn, ["TargetBoost", "BoostCurve"])
fc.add_in_measurement(fn, ["EngineSpeed", "BoostPressure", "ChargeAirTemp"])

gc = GroupCreator(session)
grp = gc.create_group("TurboParams", "Turbo parameters",
                      module_name="TCU")
gc.add_root(grp)
gc.add_ref_characteristic(grp, ["TargetBoost", "BoostCurve"])
gc.add_ref_measurement(grp, ["EngineSpeed", "BoostPressure", "ChargeAirTemp"])

# --- Commit and export ---
gc.commit()
db.close()

export_a2l("TurboECU", "TurboECU.a2l")
print("Done - exported TurboECU.a2l")

# --- Verify with inspect ---
session2 = DB().open_existing("TurboECU")
prj = Project(session2)
mod = prj.module[0]
print(f"Module: {mod.name}")
print(f"  Measurements:    {sum(1 for _ in mod.measurement.query())}")
print(f"  Characteristics: {sum(1 for _ in mod.characteristic.query())}")
print(f"  Functions:       {sum(1 for _ in mod.function.query())}")
print(f"  Groups:          {sum(1 for _ in mod.group.query())}")
```

8.4 Validate API

The validation API checks A2L databases for structural issues and common mistakes.

```python
from pya2l import DB
from pya2l.api.validate import Validator, Level, Category

session = DB().open_existing("ASAP2_Demo_V161")

validator = Validator(session)
diagnostics = validator()      # returns tuple of Message namedtuples

for msg in diagnostics:
    print(f"[{msg.type.name}] {msg.category.name}: {msg.text}")
```

8.4.1 Message fields

Each `Message` is a named tuple with:

Field	Type	Description
`type`	`Level`	`INFORMATION`, `WARNING`, or `ERROR`
`category`	`Category`	`DUPLICATE`, `MISSING`, `OBSOLETE`
`diag_code`	`Diagnostics`	Specific diagnostic code
`text`	`str`	Human-readable description

8.4.2 Diagnostic codes

Code	Meaning
`MULTIPLE_DEFINITIONS_IN_NAMESPACE`	Same name used twice in one module
`DEFINITION_IN_MULTIPLE_NAMESPACES`	Entity exists in multiple modules
`INVALID_C_IDENTIFIER`	Name is not a valid C identifier
`MISSING_BYTE_ORDER`	No byte order specified
`MISSING_ALIGNMENT`	No alignment settings in MOD_COMMON
`MISSING_EPK`	No EPK (ECU Program Key) defined
`MISSING_ADDR_EPK`	No ADDR_EPK specified
`MISSING_MODULE`	Project has no modules
`DEPRECATED`	Use of deprecated A2L features
`OVERLAPPING_MEMORY`	Memory ranges overlap

Practical validation workflow:

```python
from pya2l import DB
from pya2l.api.validate import Validator, Level

session = DB().open_existing("my_project")
diags = Validator(session)()

errors   = [d for d in diags if d.type == Level.ERROR]
warnings = [d for d in diags if d.type == Level.WARNING]
info     = [d for d in diags if d.type == Level.INFORMATION]

print(f"Errors: {len(errors)}, Warnings: {len(warnings)}, Info: {len(info)}")
```

(continues on next page)

(continued from previous page)

```
if errors:
    print("\n--- ERRORS ---")
    for e in errors:
        print(f"  [{e.diag_code.name}] {e.text}")
```

8.5 Low-Level ORM Access

For advanced queries that the inspect API doesn't cover, use SQLAlchemy directly with `pya2l.model`:

```
import pya2l.model as model
from pya2l import DB

session = DB().open_existing("ASAP2_Demo_V161")

# Raw SQL query
measurements = (
    session.query(model.Measurement)
    .filter(model.Measurement.datatype == "FLOAT32_IEEE")
    .order_by(model.Measurement.name)
    .all()
)
for m in measurements:
    print(f"{m.name}: {m.datatype} @ 0x{m.ecu_address.address:08X}")

# Projection (select specific columns)
names_and_types = (
    session.query(model.Measurement.name, model.Measurement.datatype)
    .filter(model.Measurement.name.like("ENGINE_%"))
    .all()
)

# Count
n = session.query(model.Measurement).count()

# Join across relationships
chars_with_axes = (
    session.query(model.Characteristic)
    .filter(model.Characteristic.type == "MAP")
    .all()
)

# Bridge from ORM to inspect
from pya2l.api.inspect import Measurement as MeasInspect
for row in measurements:
    meas_obj = MeasInspect.get(session, row.name)
    print(f"{meas_obj.name}: {meas_obj.compuMethod.conversionType}")
```

CHAPTER

NINE

HOW-TOS

Quick, task-oriented guides for common workflows.

9.1 Import once, reuse the database

Persist a parsed A2L as SQLite (.a2ldb) so you don't reparse on every run:

```
from pya2l import DB

db = DB()
session = db.import_a2l("ASAP2_Demo_V161.a2l")  # writes ASAP2_Demo_V161.a2ldb
```

Later, open the database without reparsing:

```
from pya2l import DB

db = DB()
session = db.open_existing("ASAP2_Demo_V161")  # .a2ldb suffix implied
```

9.2 Export back to A2L or JSON

Export preserves **all** model attributes (measurements, characteristics, conversions, metadata, IF_DATA, annotations, etc.) with no data loss.

9.2.1 Basic A2L export

Round-trip a database back to A2L text format:

```
from pya2l import export_a2l

# Export with implicit .a2ldb extension
export_a2l("ASAP2_Demo_V161", "exported.a2l")

# Or provide full database path
export_a2l("path/to/MyProject.a2ldb", "exported.a2l")

# Export to stdout (useful for piping)
import sys
from pya2l.imex.a2l_exporter import export_a2l_to_stream
```

(continues on next page)

(continued from previous page)

```
with open("MyProject.a2ldb", "rb"):
    pass  # ensure DB exists
export_a2l_to_stream("MyProject.a2ldb", sys.stdout)
```

9.2.2 JSON export

Export to JSON for external tools, scripts, or analysis:

```
from pya2l.imex.json_exporter import export_json

# Export entire database to JSON
export_json("ASAP2_Demo_V161.a2ldb", "exported.json")

# JSON structure mirrors A2L hierarchy:
# {
#   "project": { ... },
#   "modules": [
#     {
#       "name": "ModuleName",
#       "measurements": [ ... ],
#       "characteristics": [ ... ],
#       "compu_methods": [ ... ],
#       ...
#     }
#   ]
# }
```

9.2.3 Export after modifications

Typical workflow: import, modify, export:

```
from pya2l import DB, export_a2l
from pya2l.api.create import MeasurementCreator

# Import existing A2L
db = DB()
session = db.import_a2l("original.a2l")

# Add new measurement
mc = MeasurementCreator(session)
meas = mc.create_measurement(
    "NewSignal", "Added programmatically",
    "UWORD", "NO_COMPU_METHOD", 1, 0.1,
    0.0, 65535.0, module_name="MyModule"
)
mc.add_ecu_address(meas, 0x50000)
mc.commit()

db.close()

# Export modified database
export_a2l("original", "modified.a2l")
```

9.2.4 Export completeness guarantees

The exporters (as of v0.10.2+) export **all** optional model elements:

- **AXIS_PTS**: ECU_ADDRESS_EXTENSION, EXTENDED_LIMITS, FORMAT, GUARD_RAILS, MAX_REFRESH, MODEL_LINK, MONOTONY, PHYS_UNIT, SYMBOL_LINK, etc.
- **BLOB**: ADDRESS_TYPE, ANNOTATION, DISPLAY_IDENTIFIER, ECU_ADDRESS_EXTENSION, IF_DATA, MAX_REFRESH, MODEL_LINK, SYMBOL_LINK, CALIBRATION_ACCESS
- **CHARACTERISTIC**: All 20+ optional elements including CALIBRATION_ACCESS, COMPARISON_QUANTITY, DEPENDENT_CHARACTERISTIC, DISCRETE, DISPLAY_IDENTIFIER, ECU_ADDRESS_EXTENSION, ENCODING, EXTENDED_LIMITS, FORMAT, GUARD_RAILS, MODEL_LINK, NUMBER, SYMBOL_LINK, VIRTUAL_CHARACTERISTIC, axis descriptors
- **MEASUREMENT**: ADDRESS_TYPE, BIT_OPERATION (with shifts), BYTE_ORDER, DISCRETE, DISPLAY_IDENTIFIER, ECU_ADDRESS, ECU_ADDRESS_EXTENSION, ERROR_MASK, FORMAT, FUNCTION_LIST, LAYOUT, MATRIX_DIM, MAX_REFRESH, MODEL_LINK, PHYS_UNIT, SYMBOL_LINK, VIRTUAL
- **IF_DATA**: Preserved as raw blocks; custom AML parsing supported
- **Annotations**: Preserved with labels, origins, and text blocks

This ensures lossless roundtrips: `original.a2l` → import → export → `output.a2l` will preserve all content (modulo whitespace/formatting).

9.3 CLI import/export (`a2ldb-imex`)

Use the bundled console script for quick import/export tasks:

9.3.1 Basic usage

```
# Show help and available options
a2ldb-imex -h

# Show version
a2ldb-imex -V
```

9.3.2 Import examples

```
# Import A2L (creates .a2ldb next to input file)
a2ldb-imex -i path/to/file.a2l

# Import with explicit encoding
a2ldb-imex -i file.a2l -E utf-8

# Create .a2ldb in current working directory instead of next to input
a2ldb-imex -i path/to/file.a2l -L

# Silence progress bar
a2ldb-imex -i file.a2l -p

# Combine options: UTF-8 encoding, local DB, silent mode
a2ldb-imex -i examples\\ASAP2_Demo_V161.a2l -E utf-8 -L -p
```

9.3.3 Export examples

```
# Export .a2ldb back to A2L text (writes to file)
a2ldb-imex -e ASAP2_Demo_V161.a2ldb -o exported.a2l

# Export to stdout (useful for piping or inspection)
a2ldb-imex -e file.a2ldb > exported.a2l

# Export specific module (if database contains multiple)
a2ldb-imex -e file.a2ldb -m ModuleName -o module_only.a2l
```

9.3.4 JSON export

Use `--json` to export as JSON instead of A2L text:

```
# Export to JSON file
a2ldb-imex -e file.a2ldb --json -o exported.json

# Pretty-printed JSON (human-readable, larger file)
a2ldb-imex -e file.a2ldb --json --pretty -o exported.json

# JSON to stdout (for piping to jq, Python, etc.)
a2ldb-imex -e file.a2ldb --json | jq '.modules[0].measurements | length'

# Combine with module filter
a2ldb-imex -e file.a2ldb --json --pretty -m ModuleName -o module.json
```

9.3.5 Typical workflows

Validation pipeline (import, validate, re-export):

```
# Import
a2ldb-imex -i original.a2l

# (Use Python API or other tools to inspect/validate the .a2ldb)

# Re-export
a2ldb-imex -e original.a2ldb -o validated.a2l
```

Format conversion (A2L JSON):

```
# Import A2L
a2ldb-imex -i input.a2l

# Export to JSON directly via CLI
a2ldb-imex -e input.a2ldb --json -o output.json

# Pretty-printed JSON
a2ldb-imex -e input.a2ldb --json --pretty -o output.json

# Or via Python API
python -c "from pya2l.imex.json_exporter import export_json; export_json('input.a2ldb',
↪'output.json')"
```

Batch processing:

```
# Windows batch
for %%f in (*.a2l) do a2ldb-imex -i "%%f" -L

# Linux/macOS shell
for f in *.a2l; do a2ldb-imex -i "$f" -L; done
```

9.4 Concurrent access and export safety

The database uses SQLite's **WAL (Write-Ahead Logging)** mode to support concurrent readers during export operations.

9.4.1 Safe concurrent export

Multiple processes can **read/export** the same database simultaneously:

```
from pya2l import export_a2l
import multiprocessing

def export_worker(db_path, output_path):
    """Worker function for parallel exports."""
    export_a2l(db_path, output_path)

# Export the same DB to multiple formats concurrently
with multiprocessing.Pool(3) as pool:
    pool.starmap(export_worker, [
        ("project.a2ldb", "export1.a2l"),
        ("project.a2ldb", "export2.a2l"),
        ("project.a2ldb", "export3.a2l"),
    ])
```

9.4.2 Export while another process writes

Exports can run **while another process modifies** the database (though the export sees a snapshot from when it started):

```
import threading
from pya2l import DB, export_a2l
from pya2l.api.create import MeasurementCreator

def writer_task():
    """Modify database in background."""
    db = DB()
    session = db.open_existing("project")
    mc = MeasurementCreator(session)
    # ... add measurements ...
    mc.commit()
    db.close()

def export_task():
    """Export database concurrently."""
    export_a2l("project", "concurrent_export.a2l")
```

(continues on next page)

(continued from previous page)

```
# Start both tasks simultaneously
t1 = threading.Thread(target=writer_task)
t2 = threading.Thread(target=export_task)
t1.start()
t2.start()
t1.join()
t2.join()
```

The exporter sets **query_only=ON** pragma and uses a 5-second busy timeout, ensuring robust operation under concurrent load without "database is locked" errors.

9.5 Dump measurements to Excel

Use pandas to move selected fields into Excel (install `pandas` and `openpyxl` first):

```
import pandas as pd
from pya2l import DB, model

session = DB().open_existing("ASAP2_Demo_V161")
q = (
    session.query(
        model.Measurement.name,
        model.Measurement.datatype,
        model.Measurement.conversion,
        model.Measurement.ecu_address,
    )
    .order_by(model.Measurement.name)
)
df = pd.read_sql(q.statement, session.bind)
df.to_excel("measurements.xlsx", index=False)
```

9.6 Handle encodings and quiet mode

Override the default `latin-1` import encoding and silence the progress bar:

```
from pya2l import DB

db = DB()
session = db.import_a2l(
    "my_file.a2l",
    encoding="utf-8",
    progress_bar=False,
    loglevel="ERROR",  # also suppresses progress
)
```

9.7 Creating A2L content programmatically

Use the Creator API (`pya2l.api.create`) to build or augment A2L databases. All creator classes follow a consistent pattern: instantiate with a session, call `create_*` methods to add entities, then commit.

9.7.1 Basic workflow

```
from pya2l import DB
from pya2l.api.create import ProjectCreator, ModuleCreator

db = DB()
session = db.open_create("MyProject.a2ldb")

# Create project
pc = ProjectCreator(session)
project = pc.create_project("DemoProject", "Example ECU calibration")

# Create module
mc = ModuleCreator(session)
module = mc.create_module("DemoModule", "Main module", project=project)

# Commit changes
mc.commit()
db.close()
```

9.7.2 Creating conversion methods (COMPU_METHOD)

Define how raw ECU values map to physical units:

```
from pya2l.api.create import CompuMethodCreator

cmc = CompuMethodCreator(session)

# Linear conversion: phys = a*x + b
cm_linear = cmc.create_compu_method(
    "CM_Temperature", "Temperature conversion",
    "LINEAR", "%6.2", "°C", module_name="DemoModule"
)
cmc.add_coeffs_linear(cm_linear, offset=-40.0, factor=0.1)

# Rational conversion: phys = (a*x² + b*x + c) / (d*x² + e*x + f)
cm_rational = cmc.create_compu_method(
    "CM_Pressure", "Non-linear pressure", "RAT_FUNC",
    "%8.3", "bar", module_name="DemoModule"
)
cmc.add_formula_rational(cm_rational, a=0, b=0.01, c=0, d=0, e=0, f=1)

# Tabular conversion (value pairs)
cm_tab = cmc.create_compu_method(
    "CM_GearState", "Gear position", "TAB_VERB",
    "%d", "", module_name="DemoModule"
)
cmc.add_compu_tab_verbal_range(cm_tab, [
```

(continues on next page)

(continued from previous page)

```
    (0, 0, "Neutral"),
    (1, 1, "First"),
    (2, 2, "Second"),
    (3, 3, "Third"),
])
```

9.7.3 Creating measurements

Measurements are ECU signals read by calibration tools:

```
from pya2l.api.create import MeasurementCreator

mec = MeasurementCreator(session)

# Scalar measurement
meas = mec.create_measurement(
    "EngineSpeed", "Engine rotational speed",
    "UWORD", "CM_Speed", resolution=1, accuracy=0.5,
    lower_limit=0.0, upper_limit=8000.0,
    module_name="DemoModule"
)
mec.add_ecu_address(meas, 0x10000)

# Measurement with matrix dimensions (e.g., 2D sensor array)
meas_matrix = mec.create_measurement(
    "SensorArray", "Temperature sensor grid",
    "SWORD", "CM_Temperature", resolution=1, accuracy=0.1,
    lower_limit=-40.0, upper_limit=150.0,
    module_name="DemoModule"
)
mec.add_matrix_dim(meas_matrix, dims=[8, 8])  # 8×8 grid
mec.add_ecu_address(meas_matrix, 0x20000)

# Measurement with symbol link (no direct address)
meas_sym = mec.create_measurement(
    "SymLinkedSignal", "Signal via symbol",
    "FLOAT32_IEEE", "CM_Voltage", resolution=1, accuracy=0.01,
    lower_limit=0.0, upper_limit=5.0,
    module_name="DemoModule"
)
mec.add_symbol_link(meas_sym, "g_sensor_voltage", offset=0)
```

9.7.4 Creating characteristics (calibration parameters)

Characteristics are tunable parameters written by calibration tools:

```
from pya2l.api.create import CharacteristicCreator

cc = CharacteristicCreator(session)

# Scalar (VALUE) characteristic
char_value = cc.create_characteristic(
```

(continues on next page)

(continued from previous page)

```
    "InjectionDuration", "Fuel injection time",
    "VALUE", 0x30000, "RL_UWORD", 0.0, "CM_Time",
    0.0, 100.0, module_name="DemoModule"
)

# Curve (1D lookup table)
char_curve = cc.create_characteristic(
    "ThrottleCurve", "Throttle position vs. airflow",
    "CURVE", 0x31000, "RL_CURVE_8", 0.0, "CM_Airflow",
    0.0, 1000.0, module_name="DemoModule"
)
cc.add_axis_descr(char_curve, "STD_AXIS", "NO_INPUT_QUANTITY",
                  "CM_Percent", 8, 0.0, 100.0)

# Map (2D lookup table)
char_map = cc.create_characteristic(
    "InjectionMap", "RPM vs. Load injection map",
    "MAP", 0x32000, "RL_MAP_16x16", 0.0, "CM_Time",
    0.0, 50.0, module_name="DemoModule"
)
# X-axis: RPM
cc.add_axis_descr(char_map, "STD_AXIS", "EngineSpeed",
                  "CM_Speed", 16, 0.0, 8000.0)
# Y-axis: Load
cc.add_axis_descr(char_map, "STD_AXIS", "EngineLoad",
                  "CM_Percent", 16, 0.0, 100.0)
```

9.7.5 Creating axis points (shared axes)

AXIS_PTS define reusable axis definitions for multiple characteristics:

```
from pya2l.api.create import AxisPtsCreator

apc = AxisPtsCreator(session)

axis = apc.create_axis_pts(
    "RPM_Axis", "Standard RPM breakpoints",
    0x40000, "NO_INPUT_QUANTITY", "RL_AXIS_16",
    0.0, "CM_Speed", 16, 0.0, 8000.0,
    module_name="DemoModule"
)
```

9.7.6 Creating record layouts

Record layouts describe memory structures for characteristics/measurements:

```
from pya2l.api.create import RecordLayoutCreator

rlc = RecordLayoutCreator(session)

# Simple scalar layout
rl = rlc.create_record_layout("RL_UWORD", module_name="DemoModule")
```

(continues on next page)

(continued from previous page)

```
rlc.add_fnc_values(rl, position=1, datatype="UWORD", index_mode="ALTERNATE",
                   address_type="DIRECT")

# Curve layout (axis + values)
rl_curve = rlc.create_record_layout("RL_CURVE_8", module_name="DemoModule")
rlc.add_axis_pts_x(rl_curve, position=1, datatype="UWORD", index_incr="INDEX",
                   address_type="DIRECT")
rlc.add_fnc_values(rl_curve, position=2, datatype="UWORD", index_mode="ALTERNATE",
                   address_type="DIRECT")
```

9.7.7 Organizing with groups and functions

Group related entities for better organization:

```
mc = ModuleCreator(session)

# Create a function (logical grouping)
func = mc.add_function(
    module, name="InjectionControl",
    long_identifier="Fuel injection calibration parameters"
)
mc.add_def_characteristic(func, ["InjectionDuration", "InjectionMap"])
mc.add_ref_characteristic(func, ["ThrottleCurve"])  # read-only reference
mc.add_in_measurement(func, ["EngineSpeed", "EngineLoad"])

# Create a group (GUI folder structure)
grp = mc.add_group(
    module, name="EngineParams",
    long_identifier="All engine-related parameters"
)
mc.add_group_ref_characteristic(grp, ["InjectionDuration", "InjectionMap"])
mc.add_group_ref_measurement(grp, ["EngineSpeed"])
mc.add_group_sub_group(grp, ["AdvancedSettings"])  # nested groups
```

9.7.8 Adding units

Define physical units for conversions:

```
mc = ModuleCreator(session)

unit_rpm = mc.add_unit(
    module, name="rpm",
    long_identifier="Revolutions per minute",
    display="rpm", type_str="DERIVED"
)

unit_celsius = mc.add_unit(
    module, name="degC",
    long_identifier="Degrees Celsius",
    display="°C", type_str="TEMPERATURE"
)
```

9.7.9 Complete example: building a minimal ECU database

```
from pya2l import DB
from pya2l.api.create import (
    ProjectCreator, ModuleCreator, CompuMethodCreator,
    MeasurementCreator, CharacteristicCreator,
    RecordLayoutCreator
)

db = DB()
session = db.open_create("MinimalECU.a2ldb")

# Project and module
pc = ProjectCreator(session)
project = pc.create_project("MinimalECU", "Minimal ECU example")

mc = ModuleCreator(session)
module = mc.create_module("Engine", "Engine control", project=project)

# Unit
mc.add_unit(module, name="rpm", long_identifier="RPM",
            display="rpm", type_str="DERIVED")

# Conversion
cmc = CompuMethodCreator(session)
cm = cmc.create_compu_method(
    "CM_RPM", "RPM conversion", "LINEAR",
    "%8.2", "rpm", module_name="Engine"
)
cmc.add_coeffs_linear(cm, offset=0.0, factor=0.25)

# Record layout
rlc = RecordLayoutCreator(session)
rl = rlc.create_record_layout("RL_UWORD", module_name="Engine")
rlc.add_fnc_values(rl, position=1, datatype="UWORD",
                   index_mode="ALTERNATE", address_type="DIRECT")

# Measurement
mec = MeasurementCreator(session)
meas = mec.create_measurement(
    "EngineSpeed", "Current engine speed",
    "UWORD", "CM_RPM", resolution=1, accuracy=1.0,
    lower_limit=0.0, upper_limit=8000.0,
    module_name="Engine"
)
mec.add_ecu_address(meas, 0x100000)

# Characteristic
cc = CharacteristicCreator(session)
char = cc.create_characteristic(
    "IdleSpeed", "Target idle speed",
    "VALUE", 0x200000, "RL_UWORD", 0.0, "CM_RPM",
    500.0, 1500.0, module_name="Engine"
```

(continues on next page)

(continued from previous page)

```
)

# Group
grp = mc.add_group(module, name="BasicParams",
                   long_identifier="Basic parameters")
mc.add_group_ref_characteristic(grp, ["IdleSpeed"])
mc.add_group_ref_measurement(grp, ["EngineSpeed"])

# Commit and close
mc.commit()
db.close()

# Export to A2L
from pya2l import export_a2l
export_a2l("MinimalECU", "MinimalECU.a2l")
```

9.7.10 Tips for using the Creator API

- **Always commit**: Call `creator.commit()` before closing the database.
- **Module names**: Most `create_*` methods accept `module_name="..."` to associate entities with a specific module.
- **Referential integrity**: Creators validate references (e.g., conversion names, record layout names) exist before creating entities.
- **Incremental builds**: Open an existing database with `open_existing()` and add to it; useful for augmenting imported A2L files.
- **Inspect to verify**: Use `pya2l.api.inspect` classes to query and validate what you've created.

9.7.11 Available creator classes

Full list in `pya2l.api.create`:

- `ProjectCreator` – PROJECT
- `ModuleCreator` – MODULE, UNIT, GROUP, FUNCTION, FRAME, TRANSFORMER, etc.
- `CompuMethodCreator` – COMPU_METHOD, COMPU_TAB, COMPU_VTAB
- `MeasurementCreator` – MEASUREMENT
- `CharacteristicCreator` – CHARACTERISTIC
- `AxisPtsCreator` – AXIS_PTS
- `RecordLayoutCreator` – RECORD_LAYOUT

See `pya2l/examples/create_quickstart.py` for more examples.

9.8 Performance & Best Practices

pyA2L is optimized for typical A2L file sizes (<20MB) with automatic performance tuning. This section covers performance characteristics, large file handling, and best practices for optimal throughput.

9.8.1 Import performance characteristics

Measured throughput (v0.10.2+, adaptive flush strategy):

File Size	Import Time	Throughput	Peak Memory	Objects/sec
0.15 MB	4-6s	0.03 MB/s	223 MiB	~15/s
8.7 MB	38-40s	0.22 MB/s	755 MiB	~310/s
16 MB	77-80s	0.21 MB/s	1.35 GiB	~250/s
50 MB (proj.)	~250s	0.20 MB/s	~4.1 GiB	~200/s

Performance bottleneck: The C++ parser (ANTLR4-based) is very fast (2.5 MB/s), accounting for only 10% of import time. The main bottleneck is SQLAlchemy object creation and database insertion (90% of time).

Memory scaling: Memory usage scales linearly with file size for files >1MB. Small files have higher overhead due to session setup costs.

9.8.2 Handling large files (>50MB)

For very large A2L files, consider:

1. **Use ``progress_bar=False``** to avoid rendering overhead:

```
db = DB()
session = db.import_a2l("large_file.a2l", progress_bar=False)
```

2. **Import once, reuse the .a2ldb**: Avoid reparsing on every run:

```
# First run: import (slow)
db.import_a2l("large_file.a2l")  # creates large_file.a2ldb

# Subsequent runs: open existing (fast)
db.open_existing("large_file")  # instant
```

3. **Use selective queries**: Don't load entire tables into memory:

```
from pya2l import DB, model

session = DB().open_existing("large_file")

# Bad: loads all measurements into memory
all_measurements = session.query(model.Measurement).all()

# Good: iterate without loading all
for meas in session.query(model.Measurement).yield_per(1000):
    process(meas)

# Best: filter and project only needed columns
result = session.query(
    model.Measurement.name,
    model.Measurement.ecu_address
).filter(
    model.Measurement.datatype == "FLOAT32_IEEE"
).all()
```

4. **Consider JSON export for analysis**: JSON exports are faster for downstream processing:

```
from pya2l.imex.json_exporter import export_json

# Export to JSON (faster than A2L)
export_json("large_file.a2ldb", "large_file.json")

# Use standard JSON tools for processing
import json
with open("large_file.json") as f:
    data = json.load(f)
    measurements = data["modules"][0]["measurements"]
```

9.8.3 Optimizing exports

Export performance depends heavily on lazy loading behavior. Some tips:

1. **Close other database connections** before exporting:

```
from pya2l import DB, export_a2l

db = DB()
session = db.import_a2l("file.a2l")
# ... do work ...
db.close()  # Close before export!

export_a2l("file", "output.a2l")  # Faster without active sessions
```

2. **Use concurrent exports** for multiple outputs (WAL mode supports this):

```
import multiprocessing
from pya2l import export_a2l
from pya2l.imex.json_exporter import export_json

def export_a2l_worker():
    export_a2l("project", "output.a2l")

def export_json_worker():
    export_json("project.a2ldb", "output.json")

# Export both formats in parallel
p1 = multiprocessing.Process(target=export_a2l_worker)
p2 = multiprocessing.Process(target=export_json_worker)
p1.start()
p2.start()
p1.join()
p2.join()
```

3. **JSON is faster than A2L**: For data analysis, prefer JSON export (20-30% faster than A2L text generation).

9.8.4 Memory-efficient iteration

When processing large datasets, use SQLAlchemy's `yield_per()` to avoid loading everything into memory:

```
from pya2l import DB, model
```

(continues on next page)

(continued from previous page)

```
session = DB().open_existing("large_database")

# Memory-efficient: processes 1000 measurements at a time
for chunk in session.query(model.Measurement).yield_per(1000):
    for meas in chunk:
        # Process one measurement
        print(f"{meas.name}: {meas.ecu_address:#x}")

# Alternative: iterate with limit/offset for explicit pagination
page_size = 1000
offset = 0
while True:
    page = session.query(model.Measurement).limit(page_size).offset(offset).all()
    if not page:
        break
    for meas in page:
        process(meas)
    offset += page_size
```

9.8.5 Database maintenance

SQLite databases benefit from periodic optimization:

```
from pya2l import DB

db = DB()
session = db.open_existing("project")

# Reclaim unused space and optimize indexes
session.execute("VACUUM")
session.execute("ANALYZE")

db.close()
```

Run `VACUUM` after deleting many entities; run `ANALYZE` after bulk inserts to update query planner statistics.

9.8.6 Performance monitoring

Track import performance in your application:

```
import time
from pya2l import DB

start = time.perf_counter()
db = DB()
session = db.import_a2l("file.a2l", loglevel="ERROR")
elapsed = time.perf_counter() - start

file_size_mb = Path("file.a2l").stat().st_size / (1024**2)
throughput = file_size_mb / elapsed

print(f"Imported {file_size_mb:.2f} MB in {elapsed:.2f}s")
```

(continues on next page)

(continued from previous page)

```
print(f"Throughput: {throughput:.2f} MB/s")

# Count objects
num_measurements = session.query(model.Measurement).count()
num_characteristics = session.query(model.Characteristic).count()
print(f"Loaded {num_measurements} measurements, {num_characteristics} characteristics")
```

9.8.7 Future optimizations

For files >100MB, consider these approaches (not yet implemented):

- **Batch insert API**: Direct SQL generation instead of ORM (2-3x speedup expected)
- **Streaming import**: Process in chunks to limit memory (<2GB for any file size)
- **Rust-based parser**: Replace Python traverse with compiled Rust (5-10x speedup)

See the GitHub Discussions for performance-related feature requests and ongoing optimization work.

CHAPTER

TEN

FREQUENTLY ASKED QUESTIONS

10.1 Where to start?

`pya2ldb` uses a SQLite database to store your A2L files to make the contained information easily accessible for your project work. You may start with the command-line script `a2ldb-imex`. There are two use-cases:

- Read an A2L file and store it to an A2LDB (import). Use the `-i` option in this case. Optionally you may want to specify an encoding (see next question) with the `-E` option, like ascii, latin-1, utf-8, ...

```
$ a2ldb-imex -i XCPlite-0002E248-5555.A2L -E latin-1
```

- Read an A2lDB file and write A2L data (export).

```
$ a2ldb-imex -e XCPlite-0002E248-5555   # A2L data gets written to standard output.
```

File extensions can be omitted, then automatic addition happens: `.a2l` (while importing), `.a2ldb` (export). Note: Depending on your operating system, `A2L` and `a2l` may be different (Unix-like OSes)! There's also a `-h`, resp. `--help` option, giving you some more details.

10.2 While importing my A2L file I'm getting strange Unicode decode errors, what can I do?

By default `pya2ldb` does its best to guess the encoding of your A2L file (by means of chardet), but this may not work in any case. Then you need to specify an encoding:

```
from pya2l import DB

db = DB()
session = db.import_a2l("", encoding="latin-1")
```

Note: There are also two command-line utilities to play around with, `file` and `chardetect`.

In action:

```
$ file examples/tst.a2l
examples/tst.a2l: UTF-8 Unicode (with BOM) text

$ chardetect examples/tst.a2l
examples/tst.a2l: UTF-8-SIG with confidence 1.0

$ file XCPlite-0002E248-5555.A2L
```

(continues on next page)

(continued from previous page)

```
XCPlite-0002E248-5555.A2L: ASCII text, with very long lines

$ chardetect XCPlite-0002E248-5555.A2L
XCPlite-0002E248-5555.A2L: ascii with confidence 1.0
```

10.3 My A2L file includes tons of files... Do I have to copy all of them to my current working directory?

No. There's an environment variable called `ASAP_INCLUDE`, which — if present — is used to search for `/INCLUDE` files. Conventions of your operating system hold. Just like C/C++ `INCLUDE` or good old `PATH`.

10.4 How do I work with IF_DATA sections in A2L files?

IF_DATA sections contain vendor-specific information in A2L files. pyA2L parses these blocks automatically (when an AML schema is present) and wraps the result in an `IfData` dataclass with three access paths:

- `if_data.if_data_parsed` — structured dicts produced by the AML parser
- `if_data.if_data_raw` — original model objects with verbatim `.raw` text
- `if_data.flatmap` — lazily built flat index for quick tag look-ups

Using the inspect API (recommended):

```python
from pya2l import DB
from pya2l.api.inspect import Project

db = DB()
session = db.open_create("ASAP2_Demo_V161.a2l")
project = Project(session)
module = project.module[0]

# Access module IF_DATA
ifd = module.if_data
print(ifd.if_data_parsed)        # list of parsed dicts
print(len(ifd.if_data_raw))      # number of raw blocks

# Quick tag look-up
if "PROTOCOL_LAYER" in ifd.flatmap:
    print(ifd.flatmap["PROTOCOL_LAYER"])

# IF_DATA on measurements
for meas in module.measurement.query():
    if meas.if_data.if_data_parsed:
        print(meas.name, meas.if_data.if_data_parsed)

# Raw text for debugging
for raw_obj in module.if_data.if_data_raw:
    print(raw_obj.raw)
```

Manual parsing with `IfDataParser`:

```
from pya2l import DB
from pya2l.aml.ifdata_parser import IfDataParser

db = DB()
session = db.open_create("ASAP2_Demo_V161.a2l")

# Create an IF_DATA parser
ifdata_parser = IfDataParser(session)

# Parse an IF_DATA section
ifdata_text = """/begin IF_DATA XCP
/begin SEGMENT 0x01 0x02 0x00 0x00 0x00
/begin CHECKSUM XCP_ADD_44 MAX_BLOCK_SIZE 0xFFFF EXTERNAL_FUNCTION "" /end CHECKSUM
/end SEGMENT
/end IF_DATA"""

result = ifdata_parser.parse(ifdata_text)
print(result)
```

See *Working with IF_DATA* for a comprehensive guide with XCP, CCP, and KWP2000 examples.

10.5 How do I create new A2L elements programmatically?

pyA2L provides creator classes in the `pya2l.api.create` module for creating new A2L elements:

```
from pya2l import DB
from pya2l.api.create import CompuMethodCreator, MeasurementCreator

db = DB()
session = db.create("new_database")

# Create a computation method
cm_creator = CompuMethodCreator(session)
compu_method = cm_creator.create_compu_method(
    name="CM_LINEAR",
    long_identifier="Linear conversion",
    conversion_type="LINEAR",
    format_str="%.2f",
    unit="km/h"
)
cm_creator.add_coeffs_linear(compu_method, a=0.1, b=0.0)

# Create a measurement
meas_creator = MeasurementCreator(session)
measurement = meas_creator.create_measurement(
    name="ENGINE_SPEED",
    long_identifier="Engine speed",
    datatype="UWORD",
    compu_method="CM_LINEAR",
    lower_limit=0,
    upper_limit=8000,
    unit="rpm"
```

(continues on next page)

(continued from previous page)

```
)

# Commit changes
session.commit()
```

See the examples in `pya2l/examples` for a more comprehensive demonstration.

10.6 How do I filter query results when working with A2L elements?

When querying A2L elements, you can use lambda functions to filter the results:

```
from pya2l import DB
from pya2l.api.inspect import Project

db = DB()
session = db.open_create("ASAP2_Demo_V161.a2l")
project = Project(session)
module = project.module[0]

# Get all measurements with FLOAT32_IEEE data type
float_measurements = list(module.measurement.query(
    lambda x: x.datatype == "FLOAT32_IEEE"
))

# Get all characteristics with names starting with "ENGINE_"
engine_chars = list(module.characteristic.query(
    lambda x: x.name.startswith("ENGINE_")
))
```

10.7 Can SYMBOL_LINK have a missing offset?

Yes, as of v0.10.2+. The `SymbolLink.offset` attribute is **optional** and can be `None`.

Why optional? Some A2L files reference symbols without explicit offsets, relying on the symbol table alone. pyA2L now tolerates this pattern.

Behavior:

- **Creator API**: `add_symbol_link()` accepts `offset=None`
- **Exporter**: Issues a warning if offset is `None` and uses `0` as fallback to ensure valid A2L syntax
- **Inspector API**: `SymbolLink.offset` may return `None`

Example (creating a measurement with symbol link but no offset):

```
from pya2l import DB
from pya2l.api.create import MeasurementCreator

db = DB()
session = db.open_create("MyProject.a2ldb")

mc = MeasurementCreator(session)
meas = mc.create_measurement(
```

(continues on next page)

(continued from previous page)

```
    "SignalName", "Signal via symbol only",
    "FLOAT32_IEEE", "NO_COMPU_METHOD", 1, 0.01,
    0.0, 100.0, module_name="MyModule"
)
# Add symbol link WITHOUT offset
mc.add_symbol_link(meas, symbol_name="g_my_signal", offset=None)
mc.commit()
db.close()
```

Export behavior: When exporting, pyA2L logs a warning and uses `0` as the offset value to produce syntactically valid A2L:

```
WARNING: SymbolLink 'g_my_signal' missing offset; using 0 as fallback.
```

Recommendation: Provide explicit offsets when known; use `None` only when the symbol table alone is sufficient for your toolchain.

10.8 What performance can I expect for large A2L files?

Import performance (v0.10.2+, with adaptive flush optimization):

- Small files (<1 MB): ~0.03 MB/s (dominated by session setup)
- Medium files (5-10 MB): ~0.22 MB/s
- Large files (15-20 MB): ~0.21 MB/s
- Very large files (50+ MB): ~0.20 MB/s, ~4 GiB memory

Key insights:

1. **C++ parser is fast**: The ANTLR4-based C++ parser runs at 2.5 MB/s, accounting for only 10% of total import time.
2. **Python DB insertion is the bottleneck**: SQLAlchemy object creation and database insertion take 90% of the time.
3. **Adaptive flush strategy**: pyA2L automatically adjusts database flush frequency based on file size:
   - Small files (<10k keywords): flush every 100 objects
   - Medium files (10k-100k keywords): scale from 200-500 objects
   - Large files (>100k keywords): flush every 1000 or 1% of total

   This provides a 10% speedup for large files compared to fixed percentage flushing.
4. **Memory scales linearly**: For files >1MB, memory usage scales predictably (~60-70 KiB per object).

Best practices for large files:

- Import once, reuse the `.a2ldb` (opening existing DB is instant)
- Use `progress_bar=False` to avoid rendering overhead
- Use selective queries with filters instead of loading entire tables
- Consider JSON export for downstream processing (20-30% faster)

Export performance:

- A2L text export: dominated by lazy loading (93% of time)

- JSON export: 20-30% faster than A2L
- Concurrent exports supported (WAL mode allows multiple readers)

See the *HOW-TOs* section on "Performance & Best Practices" for detailed optimization techniques and code examples.

10.9 Does the exporter preserve all A2L attributes during roundtrip?

Yes, as of v0.10.2+. The A2L and JSON exporters have been systematically audited to export **all** model attributes with no data loss.

Comprehensive coverage includes:

- All optional keywords (ECU_ADDRESS_EXTENSION, EXTENDED_LIMITS, FORMAT, GUARD_RAILS, MAX_REFRESH, MODEL_LINK, SYMBOL_LINK, etc.)
- Boolean flags (DISCRETE, GUARD_RAILS, READ_ONLY, etc.)
- Complex relationships (DEPENDENT_CHARACTERISTIC, VIRTUAL_CHARACTERISTIC, COMPARISON_QUANTITY)
- Nested structures (BIT_OPERATION with left/right shifts, AXIS_DESCR, structure components)
- Annotations, IF_DATA sections, and comments

Roundtrip guarantee: `original.a2l` → import → export → `output.a2l` preserves all semantic content (whitespace and comment formatting may differ).

Testing: Use the validator to compare before/after:

```
from pya2l import DB
from pya2l.api.validate import Validator

# Import and export
db = DB()
session = db.import_a2l("original.a2l")
db.close()
from pya2l import export_a2l
export_a2l("original", "roundtrip.a2l")

# Validate both
session1 = DB().open_existing("original")
session2 = DB().import_a2l("roundtrip.a2l")

for sess in [session1, session2]:
    vd = Validator(sess)
    issues = list(vd())
    print(f"Issues found: {len(issues)}")
```

10.10 Any missing questions and answers?

There's a discussion on GitHub: https://github.com/christoph2/pyA2L/discussions/33 — feel free to ask or propose additions!

CHAPTER

ELEVEN

PYA2L

11.1 pya2l package

11.1.1 Subpackages

pya2l.api package

Submodules

pya2l.api.create module

pya2l.api.inspect module

pya2l.api.validate module

Module contents

pya2l.cgen package

Subpackages

pya2l.cgen.templates package

Module contents

Module contents

pya2l.model package

Submodules

pya2l.model.mixins module

Module contents

11.1.2 Submodules

11.1.3 pya2l.a2l module

11.1.4 pya2l.classes module

11.1.5 pya2l.exceptions module

11.1.6 pya2l.functions module

11.1.7 pya2l.logger module

11.1.8 pya2l.parserlib module

11.1.9 pya2l.templates module

11.1.10 pya2l.utils module

11.1.11 Module contents

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search