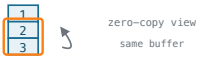
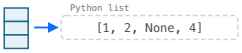
PyArrow

The essentials of PyArrow, Python's gateway to the Arrow columnar format, moving data between pandas, Polars, and Parquet at zero-copy speed.

Arrays & ChunkedArray

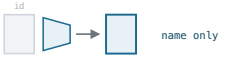
The column is the atom; Arrow arrays are typed, contiguous, and null-aware.

PURPOSE	COMMAND	RESULT
Build a typed column from a list.	`arr = pa.array([1, 2, None, 4])`	
Pin the type explicitly.	`pa.array([1, 2, 3], type=pa.int16())`	
Stitch many arrays into one column.	`ca = pa.chunked_array([[1, 2], [3, 4]])`	
Collapse chunks into one buffer.	`ca.combine_chunks()`	
Slice without copying.	`arr.slice(1, 2)`	
Read values back to Python.	`arr.to_pylist()`	

A column is a contiguous typed buffer plus a null bitmap; chunked_array grows it across buffers and slice is a zero-copy view

Table & Schema

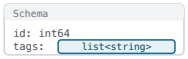
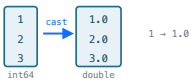
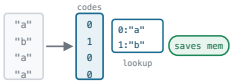
A Table is named columns plus a Schema; columns stay columnar.

PURPOSE	COMMAND	RESULT
Build a Table from a dict.	`t = pa.table({"id": [...], "name": [...]})`	
Inspect the schema.	`t.schema`	
Grab one column.	`t.column("id")`	
Add a derived column.	`t.append_column("flag", pa.array([...]))`	
Project a subset of columns.	`t.select(["name"])`	
Dump back to Python / batches.	`t.to_pydict(), t.to_batches()`	

pa.table({...}) builds named ChunkedArray columns + a Schema; select / append_column / column stay columnar and cheap

Types & Nullability

Every column has an explicit type and a per-value null bit.

PURPOSE	COMMAND	RESULT
Scalar and temporal types.	`pa.int64(), pa.string(), pa.timestamp("us")`	
Build a Schema with a list type.	`pa.schema([("id", pa.int64()), ("tags", pa.list_(pa.string()))])`	
Make a field non-nullable.	`pa.field("id", pa.int64(), nullable=False)`	
Cast a column to another type.	`arr.cast(pa.float64())`	
Dictionary-encode (categoricals).	`arr.dictionary_encode()`	
Nested struct columns.	`pa.array([{"a": 1, "b": "z"}])`	

Declare type + nullability per field in a Schema; cast changes type, dictionary_encode is Arrow's categorical, struct/list nest columns

Parquet read & write

Parquet is the columnar file format; read whole or just the columns you need.

PURPOSE	COMMAND	RESULT
Write a Table to Parquet.	`pq.write_table(t, "data.parquet")`	
Read a Parquet file back.	`pq.read_table("data.parquet")`	
Read only some columns.	`pq.read_table("data.parquet", columns=["id"])`	
Compress on write.	`pq.write_table(t, "data.parquet", compression="zstd")`	
Peek metadata without loading.	`pq.read_metadata("data.parquet").num_rows`	
Write a partitioned dataset.	`pq.write_to_dataset(t, "out/", partition_cols=["year"])`	

pq.write_table / read_table round-trip a Table; pass columns=[...] to prune, compression= to shrink, read_metadata to peek without loading

Dataset over many files

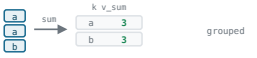
One logical table over a folder; push filters and columns down to the files.

PURPOSE	COMMAND	RESULT
Open a folder as one dataset.	`dataset = ds.dataset("out/", format="parquet")`	
Read partition keys from paths.	`ds.dataset("out/", partitioning="hive")`	
Materialize to a Table.	`dataset.to_table()`	
Push a filter down to files.	`dataset.to_table(filter=ds.field("year") == 2024)`	
Read only some columns.	`dataset.to_table(columns=["v"])`	
Stream in batches.	`dataset.scanner().head(2)`	

ds.dataset() is lazy; to_table(filter=..., columns=...) pushes predicates and projection down to skip whole partitions and unread bytes

Compute Kernels

pyarrow.compute runs vectorized C++ kernels over columns.

PURPOSE	COMMAND	RESULT
Aggregate a whole column.	`pc.sum(col), pc.mean(col)`	
Element-wise arithmetic.	`pc.multiply(col, 2)`	
Build a boolean mask.	`mask = pc.greater(col, 2)`	
Filter rows by a mask.	`t.filter(pc.equal(t["k"], "a"))`	
Group and aggregate.	`t.group_by("k").aggregate([("v", "sum")])`	
Sort the Table.	`t.sort_by([("v", "descending")])`	

pc.* are vectorized C++ kernels over columns; Table verbs filter, group_by(...).aggregate(...), and sort_by run on those same kernels

Bridge to pandas, Polars, numpy

Share the same buffers; the conversion is often zero-copy.

PURPOSE	COMMAND	RESULT
Arrow Table to a pandas DataFrame.	`df = t.to_pandas()`	
Keep Arrow-backed pandas dtypes.	`t.to_pandas(types_mapper=pd.ArrowDtype)`	
pandas DataFrame back to Arrow.	`pa.Table.from_pandas(df)`	
Hand off to / from Polars.	`pl.from_arrow(t)`, `pl_df.to_arrow()`	
Arrow array to a numpy array.	`arr.to_numpy(zero_copy_only=False)`	
numpy array to an Arrow array.	`pa.array(np.array([1.0, 2.0]))`	

Arrow is the shared layout under pandas and Polars, so handoffs reuse the same buffers; nulls into numpy force a copy

IPC & Feather

Write the in-memory layout straight to a file or stream, no re-encoding.

PURPOSE	COMMAND	RESULT
Write a Feather (IPC) file.	`feather.write_feather(t, "t.arrow")`	
Read a Feather file back.	`feather.read_table("t.arrow")`	
Write a length-prefixed IPC stream.	`with pa.ipc.new_stream(sink, t.schema) as w:` `w.write_table(t)`	
Read an IPC stream.	`with pa.ipc.open_stream(buf) as r:` `r.read_all()`	
Random-access IPC file format.	`pa.ipc.new_file(...)`, `pa.ipc.open_file(buf)`	
When to pick which.	`IPC / Feather vs Parquet`	

Feather / IPC writes the in-memory layout with almost no encode step: fastest to persist or move a Table; reach for Parquet when you want small, portable files