// Licensed to the Apache Software Foundation (ASF) under one

// or more contributor license agreements. See the NOTICE file

// distributed with this work for additional information

// regarding copyright ownership. The ASF licenses this file

// to you under the Apache License, Version 2.0 (the

// "License"); you may not use this file except in compliance

// with the License. You may obtain a copy of the License at

//

// http://www.apache.org/licenses/LICENSE-2.0

//

// Unless required by applicable law or agreed to in writing,

// software distributed under the License is distributed on an

// "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

// KIND, either express or implied. See the License for the

// specific language governing permissions and limitations

// under the License.

#include "datetime.h"

#include <algorithm>

#include <chrono>

#include <iomanip>

#include <regex>

#include <string\_view>

#include "arrow/array.h"

#include "arrow/python/arrow\_to\_python\_internal.h"

#include "arrow/python/common.h"

#include "arrow/python/helpers.h"

#include "arrow/python/platform.h"

#include "arrow/scalar.h"

#include "arrow/status.h"

#include "arrow/type.h"

#include "arrow/util/logging.h"

#include "arrow/util/regex.h"

#include "arrow/util/value\_parsing.h"

namespace arrow {

using internal::RegexMatch;

namespace py {

namespace internal {

namespace {

bool MatchFixedOffset(const std::string& tz, std::string\_view\* sign,

std::string\_view\* hour, std::string\_view\* minute) {

static const std::regex regex("^([+-])(0[0-9]|1[0-9]|2[0-3]):([0-5][0-9])$");

if (tz.size() < 5) {

return false;

}

return RegexMatch(regex, tz, {sign, hour, minute});

}

constexpr char\* NonConst(const char\* st) {

// Hack for python versions < 3.7 where members of PyStruct members

// where non-const (C++ doesn't like assigning string literals to these types)

return const\_cast<char\*>(st);

}

static PyTypeObject MonthDayNanoTupleType = {};

static PyStructSequence\_Field MonthDayNanoField[] = {

{NonConst("months"), NonConst("The number of months in the interval")},

{NonConst("days"), NonConst("The number days in the interval")},

{NonConst("nanoseconds"), NonConst("The number of nanoseconds in the interval")},

{nullptr, nullptr}};

static PyStructSequence\_Desc MonthDayNanoTupleDesc = {

NonConst("MonthDayNano"),

NonConst("A calendar interval consisting of months, days and nanoseconds."),

MonthDayNanoField,

/\*n\_in\_sequence=\*/3};

} // namespace

#ifndef PYPY\_VERSION

PyDateTime\_CAPI\* datetime\_api = nullptr;

void InitDatetime() {

PyAcquireGIL lock;

datetime\_api =

reinterpret\_cast<PyDateTime\_CAPI\*>(PyCapsule\_Import(PyDateTime\_CAPSULE\_NAME, 0));

if (datetime\_api == nullptr) {

Py\_FatalError("Could not import datetime C API");

}

}

#endif

// The following code is adapted from

// https://github.com/numpy/numpy/blob/main/numpy/core/src/multiarray/datetime.c

// Days per month, regular year and leap year

static int64\_t \_days\_per\_month\_table[2][12] = {

{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},

{31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};

static bool is\_leapyear(int64\_t year) {

return (year & 0x3) == 0 && // year % 4 == 0

((year % 100) != 0 || (year % 400) == 0);

}

// Calculates the days offset from the 1970 epoch.

static int64\_t get\_days\_from\_date(int64\_t date\_year, int64\_t date\_month,

int64\_t date\_day) {

int64\_t i, month;

int64\_t year, days = 0;

int64\_t\* month\_lengths;

year = date\_year - 1970;

days = year \* 365;

// Adjust for leap years

if (days >= 0) {

// 1968 is the closest leap year before 1970.

// Exclude the current year, so add 1.

year += 1;

// Add one day for each 4 years

days += year / 4;

// 1900 is the closest previous year divisible by 100

year += 68;

// Subtract one day for each 100 years

days -= year / 100;

// 1600 is the closest previous year divisible by 400

year += 300;

// Add one day for each 400 years

days += year / 400;

} else {

// 1972 is the closest later year after 1970.

// Include the current year, so subtract 2.

year -= 2;

// Subtract one day for each 4 years

days += year / 4;

// 2000 is the closest later year divisible by 100

year -= 28;

// Add one day for each 100 years

days -= year / 100;

// 2000 is also the closest later year divisible by 400

// Subtract one day for each 400 years

days += year / 400;

}

month\_lengths = \_days\_per\_month\_table[is\_leapyear(date\_year)];

month = date\_month - 1;

// Add the months

for (i = 0; i < month; ++i) {

days += month\_lengths[i];

}

// Add the days

days += date\_day - 1;

return days;

}

// Modifies '\*days\_' to be the day offset within the year,

// and returns the year.

static int64\_t days\_to\_yearsdays(int64\_t\* days\_) {

const int64\_t days\_per\_400years = (400 \* 365 + 100 - 4 + 1);

// Adjust so it's relative to the year 2000 (divisible by 400)

int64\_t days = (\*days\_) - (365 \* 30 + 7);

int64\_t year;

// Break down the 400 year cycle to get the year and day within the year

if (days >= 0) {

year = 400 \* (days / days\_per\_400years);

days = days % days\_per\_400years;

} else {

year = 400 \* ((days - (days\_per\_400years - 1)) / days\_per\_400years);

days = days % days\_per\_400years;

if (days < 0) {

days += days\_per\_400years;

}

}

// Work out the year/day within the 400 year cycle

if (days >= 366) {

year += 100 \* ((days - 1) / (100 \* 365 + 25 - 1));

days = (days - 1) % (100 \* 365 + 25 - 1);

if (days >= 365) {

year += 4 \* ((days + 1) / (4 \* 365 + 1));

days = (days + 1) % (4 \* 365 + 1);

if (days >= 366) {

year += (days - 1) / 365;

days = (days - 1) % 365;

}

}

}

\*days\_ = days;

return year + 2000;

}

// Extracts the month and year and day number from a number of days

static void get\_date\_from\_days(int64\_t days, int64\_t\* date\_year, int64\_t\* date\_month,

int64\_t\* date\_day) {

int64\_t \*month\_lengths, i;

\*date\_year = days\_to\_yearsdays(&days);

month\_lengths = \_days\_per\_month\_table[is\_leapyear(\*date\_year)];

for (i = 0; i < 12; ++i) {

if (days < month\_lengths[i]) {

\*date\_month = i + 1;

\*date\_day = days + 1;

return;

} else {

days -= month\_lengths[i];

}

}

// Should never get here

return;

}

// Splitting time quantities, for example splitting total seconds into

// minutes and remaining seconds. After we run

// int64\_t remaining = split\_time(total, quotient, &next)

// we have

// total = next \* quotient + remaining. Handles negative values by propagating

// them: If total is negative, next will be negative and remaining will

// always be non-negative.

static inline int64\_t split\_time(int64\_t total, int64\_t quotient, int64\_t\* next) {

int64\_t r = total % quotient;

if (r < 0) {

\*next = total / quotient - 1;

return r + quotient;

} else {

\*next = total / quotient;

return r;

}

}

static inline Status PyTime\_convert\_int(int64\_t val, const TimeUnit::type unit,

int64\_t\* hour, int64\_t\* minute, int64\_t\* second,

int64\_t\* microsecond) {

switch (unit) {

case TimeUnit::NANO:

if (val % 1000 != 0) {

return Status::Invalid("Value ", val, " has non-zero nanoseconds");

}

val /= 1000;

// fall through

case TimeUnit::MICRO:

\*microsecond = split\_time(val, 1000000LL, &val);

\*second = split\_time(val, 60, &val);

\*minute = split\_time(val, 60, hour);

break;

case TimeUnit::MILLI:

\*microsecond = split\_time(val, 1000, &val) \* 1000;

// fall through

case TimeUnit::SECOND:

\*second = split\_time(val, 60, &val);

\*minute = split\_time(val, 60, hour);

break;

default:

break;

}

return Status::OK();

}

static inline Status PyDate\_convert\_int(int64\_t val, const DateUnit unit, int64\_t\* year,

int64\_t\* month, int64\_t\* day) {

switch (unit) {

case DateUnit::MILLI:

val /= 86400000LL; // fall through

case DateUnit::DAY:

get\_date\_from\_days(val, year, month, day);

default:

break;

}

return Status::OK();

}

PyObject\* NewMonthDayNanoTupleType() {

if (MonthDayNanoTupleType.tp\_name == nullptr) {

if (PyStructSequence\_InitType2(&MonthDayNanoTupleType, &MonthDayNanoTupleDesc) != 0) {

Py\_FatalError("Could not initialize MonthDayNanoTuple");

}

}

Py\_INCREF(&MonthDayNanoTupleType);

return (PyObject\*)&MonthDayNanoTupleType;

}

Status PyTime\_from\_int(int64\_t val, const TimeUnit::type unit, PyObject\*\* out) {

int64\_t hour = 0, minute = 0, second = 0, microsecond = 0;

RETURN\_NOT\_OK(PyTime\_convert\_int(val, unit, &hour, &minute, &second, &microsecond));

\*out = PyTime\_FromTime(static\_cast<int32\_t>(hour), static\_cast<int32\_t>(minute),

static\_cast<int32\_t>(second), static\_cast<int32\_t>(microsecond));

return Status::OK();

}

Status PyDate\_from\_int(int64\_t val, const DateUnit unit, PyObject\*\* out) {

int64\_t year = 0, month = 0, day = 0;

RETURN\_NOT\_OK(PyDate\_convert\_int(val, unit, &year, &month, &day));

\*out = PyDate\_FromDate(static\_cast<int32\_t>(year), static\_cast<int32\_t>(month),

static\_cast<int32\_t>(day));

return Status::OK();

}

Status PyDateTime\_from\_int(int64\_t val, const TimeUnit::type unit, PyObject\*\* out) {

int64\_t hour = 0, minute = 0, second = 0, microsecond = 0;

RETURN\_NOT\_OK(PyTime\_convert\_int(val, unit, &hour, &minute, &second, &microsecond));

int64\_t total\_days = 0;

hour = split\_time(hour, 24, &total\_days);

int64\_t year = 0, month = 0, day = 0;

get\_date\_from\_days(total\_days, &year, &month, &day);

\*out = PyDateTime\_FromDateAndTime(

static\_cast<int32\_t>(year), static\_cast<int32\_t>(month), static\_cast<int32\_t>(day),

static\_cast<int32\_t>(hour), static\_cast<int32\_t>(minute),

static\_cast<int32\_t>(second), static\_cast<int32\_t>(microsecond));

return Status::OK();

}

int64\_t PyDate\_to\_days(PyDateTime\_Date\* pydate) {

return get\_days\_from\_date(PyDateTime\_GET\_YEAR(pydate), PyDateTime\_GET\_MONTH(pydate),

PyDateTime\_GET\_DAY(pydate));

}

Result<int64\_t> PyDateTime\_utcoffset\_s(PyObject\* obj) {

// calculate offset from UTC timezone in seconds

// supports only PyDateTime\_DateTime and PyDateTime\_Time objects

OwnedRef pyoffset(PyObject\_CallMethod(obj, "utcoffset", NULL));

RETURN\_IF\_PYERROR();

if (pyoffset.obj() != nullptr && pyoffset.obj() != Py\_None) {

auto delta = reinterpret\_cast<PyDateTime\_Delta\*>(pyoffset.obj());

return internal::PyDelta\_to\_s(delta);

} else {

return 0;

}

}

Result<std::string> PyTZInfo\_utcoffset\_hhmm(PyObject\* pytzinfo) {

// attempt to convert timezone offset objects to "+/-{hh}:{mm}" format

OwnedRef pydelta\_object(PyObject\_CallMethod(pytzinfo, "utcoffset", "O", Py\_None));

RETURN\_IF\_PYERROR();

if (!PyDelta\_Check(pydelta\_object.obj())) {

return Status::Invalid(

"Object returned by tzinfo.utcoffset(None) is not an instance of "

"datetime.timedelta");

}

auto pydelta = reinterpret\_cast<PyDateTime\_Delta\*>(pydelta\_object.obj());

// retrieve the offset as seconds

auto total\_seconds = internal::PyDelta\_to\_s(pydelta);

// determine whether the offset is positive or negative

auto sign = (total\_seconds < 0) ? "-" : "+";

total\_seconds = abs(total\_seconds);

// calculate offset components

int64\_t hours, minutes, seconds;

seconds = split\_time(total\_seconds, 60, &minutes);

minutes = split\_time(minutes, 60, &hours);

if (seconds > 0) {

// check there are no remaining seconds

return Status::Invalid("Offset must represent whole number of minutes");

}

// construct the timezone string

std::stringstream stream;

stream << sign << std::setfill('0') << std::setw(2) << hours << ":" << std::setfill('0')

<< std::setw(2) << minutes;

return stream.str();

}

// Converted from python. See https://github.com/apache/arrow/pull/7604

// for details.

Result<PyObject\*> StringToTzinfo(const std::string& tz) {

std::string\_view sign\_str, hour\_str, minute\_str;

OwnedRef pytz;

OwnedRef zoneinfo;

OwnedRef datetime;

if (internal::ImportModule("pytz", &pytz).ok()) {

if (MatchFixedOffset(tz, &sign\_str, &hour\_str, &minute\_str)) {

int sign = -1;

if (sign\_str == "+") {

sign = 1;

}

OwnedRef fixed\_offset;

RETURN\_NOT\_OK(internal::ImportFromModule(pytz.obj(), "FixedOffset", &fixed\_offset));

uint32\_t minutes, hours;

if (!::arrow::internal::ParseUnsigned(hour\_str.data(), hour\_str.size(), &hours) ||

!::arrow::internal::ParseUnsigned(minute\_str.data(), minute\_str.size(),

&minutes)) {

return Status::Invalid("Invalid timezone: ", tz);

}

OwnedRef total\_minutes(PyLong\_FromLong(

sign \* ((static\_cast<int>(hours) \* 60) + static\_cast<int>(minutes))));

RETURN\_IF\_PYERROR();

auto tzinfo =

PyObject\_CallFunctionObjArgs(fixed\_offset.obj(), total\_minutes.obj(), NULL);

RETURN\_IF\_PYERROR();

return tzinfo;

}

OwnedRef timezone;

RETURN\_NOT\_OK(internal::ImportFromModule(pytz.obj(), "timezone", &timezone));

OwnedRef py\_tz\_string(

PyUnicode\_FromStringAndSize(tz.c\_str(), static\_cast<Py\_ssize\_t>(tz.size())));

auto tzinfo = PyObject\_CallFunctionObjArgs(timezone.obj(), py\_tz\_string.obj(), NULL);

RETURN\_IF\_PYERROR();

return tzinfo;

}

// catch fixed offset if pytz is not present

if (MatchFixedOffset(tz, &sign\_str, &hour\_str, &minute\_str)) {

RETURN\_NOT\_OK(internal::ImportModule("datetime", &datetime));

int sign = -1;

if (sign\_str == "+") {

sign = 1;

}

// import timezone and timedelta module to create a tzinfo object

OwnedRef class\_timezone;

OwnedRef class\_timedelta;

RETURN\_NOT\_OK(

internal::ImportFromModule(datetime.obj(), "timezone", &class\_timezone));

RETURN\_NOT\_OK(

internal::ImportFromModule(datetime.obj(), "timedelta", &class\_timedelta));

// check input

uint32\_t minutes, hours;

if (!::arrow::internal::ParseUnsigned(hour\_str.data(), hour\_str.size(), &hours) ||

!::arrow::internal::ParseUnsigned(minute\_str.data(), minute\_str.size(),

&minutes)) {

return Status::Invalid("Invalid timezone: ", tz);

}

// save offset as a signed integer

OwnedRef total\_minutes(PyLong\_FromLong(

sign \* ((static\_cast<int>(hours) \* 60) + static\_cast<int>(minutes))));

// create zero integers for empty arguments in datetime.timedelta

OwnedRef zero(PyLong\_FromLong(static\_cast<int>(0)));

// call datetime.timedelta to get correct offset object for datetime.timezone

auto offset =

PyObject\_CallFunctionObjArgs(class\_timedelta.obj(), zero.obj(), zero.obj(),

zero.obj(), zero.obj(), total\_minutes.obj(), NULL);

RETURN\_IF\_PYERROR();

// call datetime.timezone

auto tzinfo = PyObject\_CallFunctionObjArgs(class\_timezone.obj(), offset, NULL);

RETURN\_IF\_PYERROR();

return tzinfo;

}

// fallback on zoneinfo if tz is string and pytz is not present

if (internal::ImportModule("zoneinfo", &zoneinfo).ok()) {

OwnedRef class\_zoneinfo;

RETURN\_NOT\_OK(

internal::ImportFromModule(zoneinfo.obj(), "ZoneInfo", &class\_zoneinfo));

OwnedRef py\_tz\_string(

PyUnicode\_FromStringAndSize(tz.c\_str(), static\_cast<Py\_ssize\_t>(tz.size())));

auto tzinfo =

PyObject\_CallFunctionObjArgs(class\_zoneinfo.obj(), py\_tz\_string.obj(), NULL);

RETURN\_IF\_PYERROR();

return tzinfo;

}

return Status::Invalid(

"Pytz package or Python>=3.8 for zoneinfo module must be installed.");

}

Result<std::string> TzinfoToString(PyObject\* tzinfo) {

OwnedRef module\_pytz; // import pytz

OwnedRef module\_datetime; // import datetime

OwnedRef module\_zoneinfo; // import zoneinfo

OwnedRef module\_dateutil; // import dateutil

OwnedRef class\_timezone; // from datetime import timezone

OwnedRef class\_fixedoffset; // from pytz import \_FixedOffset

OwnedRef class\_basetzinfo; // from pytz import BaseTzInfo

OwnedRef class\_zoneinfo; // from zoneinfo import ZoneInfo

OwnedRef class\_tzfile; // from zoneinfo import tzfile

// import necessary modules

RETURN\_NOT\_OK(internal::ImportModule("datetime", &module\_datetime));

// import necessary classes

RETURN\_NOT\_OK(

internal::ImportFromModule(module\_datetime.obj(), "timezone", &class\_timezone));

// check that it's a valid tzinfo object

if (!PyTZInfo\_Check(tzinfo)) {

return Status::TypeError("Not an instance of datetime.tzinfo");

}

// if tzinfo is an instance of datetime.timezone return the

// HH:MM offset string representation

if (PyObject\_IsInstance(tzinfo, class\_timezone.obj())) {

// still recognize datetime.timezone.utc as UTC (instead of +00:00)

OwnedRef tzname\_object(PyObject\_CallMethod(tzinfo, "tzname", "O", Py\_None));

RETURN\_IF\_PYERROR();

if (PyUnicode\_Check(tzname\_object.obj())) {

std::string result;

RETURN\_NOT\_OK(internal::PyUnicode\_AsStdString(tzname\_object.obj(), &result));

if (result == "UTC") {

return result;

}

}

return PyTZInfo\_utcoffset\_hhmm(tzinfo);

}

// Try to import pytz if it is available

if (internal::ImportModule("pytz", &module\_pytz).ok()) {

RETURN\_NOT\_OK(internal::ImportFromModule(module\_pytz.obj(), "\_FixedOffset",

&class\_fixedoffset));

RETURN\_NOT\_OK(

internal::ImportFromModule(module\_pytz.obj(), "BaseTzInfo", &class\_basetzinfo));

}

// if tzinfo is an instance of pytz.\_FixedOffset return the

// HH:MM offset string representation

if (module\_pytz.obj() != nullptr &&

PyObject\_IsInstance(tzinfo, class\_fixedoffset.obj())) {

OwnedRef tzname\_object(PyObject\_CallMethod(tzinfo, "tzname", "O", Py\_None));

RETURN\_IF\_PYERROR();

return PyTZInfo\_utcoffset\_hhmm(tzinfo);

}

// if pytz is installed and tzinfo is and instance of pytz.BaseTzInfo

if (module\_pytz.obj() != nullptr &&

PyObject\_IsInstance(tzinfo, class\_basetzinfo.obj())) {

OwnedRef zone(PyObject\_GetAttrString(tzinfo, "zone"));

RETURN\_IF\_PYERROR();

std::string result;

RETURN\_NOT\_OK(internal::PyUnicode\_AsStdString(zone.obj(), &result));

return result;

}

// Try to import zoneinfo if it is available

if (internal::ImportModule("zoneinfo", &module\_zoneinfo).ok()) {

RETURN\_NOT\_OK(

internal::ImportFromModule(module\_zoneinfo.obj(), "ZoneInfo", &class\_zoneinfo));

}

// if zoneinfo is installed and tzinfo is an instance of zoneinfo.ZoneInfo

if (module\_zoneinfo.obj() != nullptr &&

PyObject\_IsInstance(tzinfo, class\_zoneinfo.obj())) {

OwnedRef key(PyObject\_GetAttrString(tzinfo, "key"));

RETURN\_IF\_PYERROR();

std::string result;

RETURN\_NOT\_OK(internal::PyUnicode\_AsStdString(key.obj(), &result));

return result;

}

// Try to import dateutil if it is available

if (internal::ImportModule("dateutil.tz", &module\_dateutil).ok()) {

RETURN\_NOT\_OK(

internal::ImportFromModule(module\_dateutil.obj(), "tzfile", &class\_tzfile));

}

// if dateutil is installed and tzinfo is an instance of dateutil.tz.tzfile

if (module\_dateutil.obj() != nullptr &&

PyObject\_IsInstance(tzinfo, class\_tzfile.obj())) {

OwnedRef \_filename(PyObject\_GetAttrString(tzinfo, "\_filename"));

RETURN\_IF\_PYERROR();

std::string result;

RETURN\_NOT\_OK(internal::PyUnicode\_AsStdString(\_filename.obj(), &result));

// \_filename returns a full path in general ('/usr/share/zoneinfo/Europe/Paris')

// or POSIX name on Windows ('Europe/Paris') - we need a substring in first case

std::size\_t pos = result.find("zoneinfo/");

if (pos != std::string::npos) {

return result.substr(pos + 9);

}

return result;

}

// attempt to call tzinfo.tzname(None)

OwnedRef tzname\_object(PyObject\_CallMethod(tzinfo, "tzname", "O", Py\_None));

RETURN\_IF\_PYERROR();

if (PyUnicode\_Check(tzname\_object.obj())) {

std::string result;

RETURN\_NOT\_OK(internal::PyUnicode\_AsStdString(tzname\_object.obj(), &result));

return result;

}

// fall back to HH:MM offset string representation based on tzinfo.utcoffset(None)

return PyTZInfo\_utcoffset\_hhmm(tzinfo);

}

PyObject\* MonthDayNanoIntervalToNamedTuple(

const MonthDayNanoIntervalType::MonthDayNanos& interval) {

OwnedRef tuple(PyStructSequence\_New(&MonthDayNanoTupleType));

if (ARROW\_PREDICT\_FALSE(tuple.obj() == nullptr)) {

return nullptr;

}

PyStructSequence\_SetItem(tuple.obj(), /\*pos=\*/0, PyLong\_FromLong(interval.months));

PyStructSequence\_SetItem(tuple.obj(), /\*pos=\*/1, PyLong\_FromLong(interval.days));

PyStructSequence\_SetItem(tuple.obj(), /\*pos=\*/2,

PyLong\_FromLongLong(interval.nanoseconds));

return tuple.detach();

}

namespace {

// Wrapper around a Python list object that mimics dereference and assignment

// operations.

struct PyListAssigner {

public:

explicit PyListAssigner(PyObject\* list) : list\_(list) { DCHECK(PyList\_Check(list\_)); }

PyListAssigner& operator\*() { return \*this; }

void operator=(PyObject\* obj) {

if (ARROW\_PREDICT\_FALSE(PyList\_SetItem(list\_, current\_index\_, obj) == -1)) {

Py\_FatalError("list did not have the correct preallocated size.");

}

}

PyListAssigner& operator++() {

current\_index\_++;

return \*this;

}

PyListAssigner& operator+=(int64\_t offset) {

current\_index\_ += offset;

return \*this;

}

private:

PyObject\* list\_;

int64\_t current\_index\_ = 0;

};

} // namespace

Result<PyObject\*> MonthDayNanoIntervalArrayToPyList(

const MonthDayNanoIntervalArray& array) {

OwnedRef out\_list(PyList\_New(array.length()));

RETURN\_IF\_PYERROR();

PyListAssigner out\_objects(out\_list.obj());

auto& interval\_array =

arrow::internal::checked\_cast<const MonthDayNanoIntervalArray&>(array);

RETURN\_NOT\_OK(internal::WriteArrayObjects(

interval\_array,

[&](const MonthDayNanoIntervalType::MonthDayNanos& interval, PyListAssigner& out) {

PyObject\* tuple = internal::MonthDayNanoIntervalToNamedTuple(interval);

if (ARROW\_PREDICT\_FALSE(tuple == nullptr)) {

RETURN\_IF\_PYERROR();

}

\*out = tuple;

return Status::OK();

},

out\_objects));

return out\_list.detach();

}

Result<PyObject\*> MonthDayNanoIntervalScalarToPyObject(

const MonthDayNanoIntervalScalar& scalar) {

if (scalar.is\_valid) {

return internal::MonthDayNanoIntervalToNamedTuple(scalar.value);

} else {

Py\_INCREF(Py\_None);

return Py\_None;

}

}

} // namespace internal

} // namespace py

} // namespace arrow