// 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 "arrow/python/python\_to\_arrow.h"

#include "arrow/python/numpy\_interop.h"

#include <datetime.h>

#include <algorithm>

#include <limits>

#include <sstream>

#include <string>

#include <utility>

#include <vector>

#include "arrow/array.h"

#include "arrow/array/builder\_base.h"

#include "arrow/array/builder\_binary.h"

#include "arrow/array/builder\_decimal.h"

#include "arrow/array/builder\_dict.h"

#include "arrow/array/builder\_nested.h"

#include "arrow/array/builder\_primitive.h"

#include "arrow/array/builder\_time.h"

#include "arrow/chunked\_array.h"

#include "arrow/result.h"

#include "arrow/scalar.h"

#include "arrow/status.h"

#include "arrow/type.h"

#include "arrow/type\_traits.h"

#include "arrow/util/checked\_cast.h"

#include "arrow/util/converter.h"

#include "arrow/util/decimal.h"

#include "arrow/util/int\_util\_overflow.h"

#include "arrow/util/logging.h"

#include "arrow/python/datetime.h"

#include "arrow/python/decimal.h"

#include "arrow/python/helpers.h"

#include "arrow/python/inference.h"

#include "arrow/python/iterators.h"

#include "arrow/python/numpy\_convert.h"

#include "arrow/python/type\_traits.h"

#include "arrow/python/vendored/pythoncapi\_compat.h"

#include "arrow/visit\_type\_inline.h"

namespace arrow {

using internal::checked\_cast;

using internal::checked\_pointer\_cast;

using internal::Converter;

using internal::DictionaryConverter;

using internal::ListConverter;

using internal::PrimitiveConverter;

using internal::StructConverter;

using internal::MakeChunker;

using internal::MakeConverter;

namespace py {

namespace {

enum class MonthDayNanoField { kMonths, kWeeksAndDays, kDaysOnly, kNanoseconds };

template <MonthDayNanoField field>

struct MonthDayNanoTraits;

struct MonthDayNanoAttrData {

const char\* name;

const int64\_t multiplier;

};

template <>

struct MonthDayNanoTraits<MonthDayNanoField::kMonths> {

using c\_type = int32\_t;

static const MonthDayNanoAttrData attrs[];

};

const MonthDayNanoAttrData MonthDayNanoTraits<MonthDayNanoField::kMonths>::attrs[] = {

{"years", 1}, {"months", /\*months\_in\_year=\*/12}, {nullptr, 0}};

template <>

struct MonthDayNanoTraits<MonthDayNanoField::kWeeksAndDays> {

using c\_type = int32\_t;

static const MonthDayNanoAttrData attrs[];

};

const MonthDayNanoAttrData MonthDayNanoTraits<MonthDayNanoField::kWeeksAndDays>::attrs[] =

{{"weeks", 1}, {"days", /\*days\_in\_week=\*/7}, {nullptr, 0}};

template <>

struct MonthDayNanoTraits<MonthDayNanoField::kDaysOnly> {

using c\_type = int32\_t;

static const MonthDayNanoAttrData attrs[];

};

const MonthDayNanoAttrData MonthDayNanoTraits<MonthDayNanoField::kDaysOnly>::attrs[] = {

{"days", 1}, {nullptr, 0}};

template <>

struct MonthDayNanoTraits<MonthDayNanoField::kNanoseconds> {

using c\_type = int64\_t;

static const MonthDayNanoAttrData attrs[];

};

const MonthDayNanoAttrData MonthDayNanoTraits<MonthDayNanoField::kNanoseconds>::attrs[] =

{{"hours", 1},

{"minutes", /\*minutes\_in\_hours=\*/60},

{"seconds", /\*seconds\_in\_minute=\*/60},

{"milliseconds", /\*milliseconds\_in\_seconds\*/ 1000},

{"microseconds", /\*microseconds\_in\_milliseconds=\*/1000},

{"nanoseconds", /\*nanoseconds\_in\_microseconds=\*/1000},

{nullptr, 0}};

template <MonthDayNanoField field>

struct PopulateMonthDayNano {

using Traits = MonthDayNanoTraits<field>;

using field\_c\_type = typename Traits::c\_type;

static Status Field(PyObject\* obj, field\_c\_type\* out, bool\* found\_attrs) {

\*out = 0;

for (const MonthDayNanoAttrData\* attr = &Traits::attrs[0]; attr->multiplier != 0;

++attr) {

if (attr->multiplier != 1 &&

::arrow::internal::MultiplyWithOverflow(

static\_cast<field\_c\_type>(attr->multiplier), \*out, out)) {

return Status::Invalid("Overflow on: ", (attr - 1)->name,

" for: ", internal::PyObject\_StdStringRepr(obj));

}

OwnedRef field\_value(PyObject\_GetAttrString(obj, attr->name));

if (field\_value.obj() == nullptr) {

// No attribute present, skip to the next one.

PyErr\_Clear();

continue;

}

RETURN\_IF\_PYERROR();

\*found\_attrs = true;

field\_c\_type value;

RETURN\_NOT\_OK(internal::CIntFromPython(field\_value.obj(), &value, attr->name));

if (::arrow::internal::AddWithOverflow(\*out, value, out)) {

return Status::Invalid("Overflow on: ", attr->name,

" for: ", internal::PyObject\_StdStringRepr(obj));

}

}

return Status::OK();

}

};

// Utility for converting single python objects to their intermediate C representations

// which can be fed to the typed builders

class PyValue {

public:

// Type aliases for shorter signature definitions

using I = PyObject\*;

using O = PyConversionOptions;

// Used for null checking before actually converting the values

static bool IsNull(const O& options, I obj) {

if (options.from\_pandas) {

return internal::PandasObjectIsNull(obj);

} else {

return obj == Py\_None;

}

}

// Used for post-conversion numpy NaT sentinel checking

static bool IsNaT(const TimestampType\*, int64\_t value) {

return internal::npy\_traits<NPY\_DATETIME>::isnull(value);

}

// Used for post-conversion numpy NaT sentinel checking

static bool IsNaT(const DurationType\*, int64\_t value) {

return internal::npy\_traits<NPY\_TIMEDELTA>::isnull(value);

}

static Result<std::nullptr\_t> Convert(const NullType\*, const O&, I obj) {

if (obj == Py\_None) {

return nullptr;

} else {

return Status::Invalid("Invalid null value");

}

}

static Result<bool> Convert(const BooleanType\*, const O&, I obj) {

if (obj == Py\_True) {

return true;

} else if (obj == Py\_False) {

return false;

} else if (has\_numpy() && PyArray\_IsScalar(obj, Bool)) {

return reinterpret\_cast<PyBoolScalarObject\*>(obj)->obval == NPY\_TRUE;

} else {

return internal::InvalidValue(obj, "tried to convert to boolean");

}

}

template <typename T>

static enable\_if\_integer<T, Result<typename T::c\_type>> Convert(const T\* type, const O&,

I obj) {

typename T::c\_type value;

auto status = internal::CIntFromPython(obj, &value);

if (ARROW\_PREDICT\_TRUE(status.ok())) {

return value;

} else if (!internal::PyIntScalar\_Check(obj)) {

std::stringstream ss;

ss << "tried to convert to " << type->ToString();

return internal::InvalidValue(obj, ss.str());

} else {

return status;

}

}

static Result<uint16\_t> Convert(const HalfFloatType\*, const O&, I obj) {

uint16\_t value;

RETURN\_NOT\_OK(PyFloat\_AsHalf(obj, &value));

return value;

}

static Result<float> Convert(const FloatType\*, const O&, I obj) {

float value;

if (internal::PyFloatScalar\_Check(obj)) {

value = static\_cast<float>(PyFloat\_AsDouble(obj));

RETURN\_IF\_PYERROR();

} else if (internal::PyIntScalar\_Check(obj)) {

RETURN\_NOT\_OK(internal::IntegerScalarToFloat32Safe(obj, &value));

} else {

return internal::InvalidValue(obj, "tried to convert to float32");

}

return value;

}

static Result<double> Convert(const DoubleType\*, const O&, I obj) {

double value;

if (PyFloat\_Check(obj)) {

value = PyFloat\_AS\_DOUBLE(obj);

} else if (internal::PyFloatScalar\_Check(obj)) {

// Other kinds of float-y things

value = PyFloat\_AsDouble(obj);

RETURN\_IF\_PYERROR();

} else if (internal::PyIntScalar\_Check(obj)) {

RETURN\_NOT\_OK(internal::IntegerScalarToDoubleSafe(obj, &value));

} else {

return internal::InvalidValue(obj, "tried to convert to double");

}

return value;

}

static Result<Decimal128> Convert(const Decimal128Type\* type, const O&, I obj) {

Decimal128 value;

RETURN\_NOT\_OK(internal::DecimalFromPyObject(obj, \*type, &value));

return value;

}

static Result<Decimal256> Convert(const Decimal256Type\* type, const O&, I obj) {

Decimal256 value;

RETURN\_NOT\_OK(internal::DecimalFromPyObject(obj, \*type, &value));

return value;

}

static Result<int32\_t> Convert(const Date32Type\*, const O&, I obj) {

int32\_t value;

if (PyDate\_Check(obj)) {

auto pydate = reinterpret\_cast<PyDateTime\_Date\*>(obj);

value = static\_cast<int32\_t>(internal::PyDate\_to\_days(pydate));

} else {

RETURN\_NOT\_OK(

internal::CIntFromPython(obj, &value, "Integer too large for date32"));

}

return value;

}

static Result<int64\_t> Convert(const Date64Type\*, const O&, I obj) {

int64\_t value;

if (PyDateTime\_Check(obj)) {

auto pydate = reinterpret\_cast<PyDateTime\_DateTime\*>(obj);

value = internal::PyDateTime\_to\_ms(pydate);

// Truncate any intraday milliseconds

// TODO: introduce an option for this

value -= value % 86400000LL;

} else if (PyDate\_Check(obj)) {

auto pydate = reinterpret\_cast<PyDateTime\_Date\*>(obj);

value = internal::PyDate\_to\_ms(pydate);

} else {

RETURN\_NOT\_OK(

internal::CIntFromPython(obj, &value, "Integer too large for date64"));

}

return value;

}

static Result<int32\_t> Convert(const Time32Type\* type, const O&, I obj) {

int32\_t value;

if (PyTime\_Check(obj)) {

switch (type->unit()) {

case TimeUnit::SECOND:

value = static\_cast<int32\_t>(internal::PyTime\_to\_s(obj));

break;

case TimeUnit::MILLI:

value = static\_cast<int32\_t>(internal::PyTime\_to\_ms(obj));

break;

default:

return Status::UnknownError("Invalid time unit");

}

} else {

RETURN\_NOT\_OK(internal::CIntFromPython(obj, &value, "Integer too large for int32"));

}

return value;

}

static Result<int64\_t> Convert(const Time64Type\* type, const O&, I obj) {

int64\_t value;

if (PyTime\_Check(obj)) {

switch (type->unit()) {

case TimeUnit::MICRO:

value = internal::PyTime\_to\_us(obj);

break;

case TimeUnit::NANO:

value = internal::PyTime\_to\_ns(obj);

break;

default:

return Status::UnknownError("Invalid time unit");

}

} else {

RETURN\_NOT\_OK(internal::CIntFromPython(obj, &value, "Integer too large for int64"));

}

return value;

}

static Result<int64\_t> Convert(const TimestampType\* type, const O& options, I obj) {

int64\_t value, offset;

if (PyDateTime\_Check(obj)) {

if (ARROW\_PREDICT\_FALSE(options.ignore\_timezone)) {

offset = 0;

} else {

ARROW\_ASSIGN\_OR\_RAISE(offset, internal::PyDateTime\_utcoffset\_s(obj));

}

auto dt = reinterpret\_cast<PyDateTime\_DateTime\*>(obj);

switch (type->unit()) {

case TimeUnit::SECOND:

value = internal::PyDateTime\_to\_s(dt) - offset;

break;

case TimeUnit::MILLI:

value = internal::PyDateTime\_to\_ms(dt) - offset \* 1000LL;

break;

case TimeUnit::MICRO:

value = internal::PyDateTime\_to\_us(dt) - offset \* 1000000LL;

break;

case TimeUnit::NANO:

if (internal::IsPandasTimestamp(obj)) {

// pd.Timestamp value attribute contains the offset from unix epoch

// so no adjustment for timezone is need.

OwnedRef nanos(PyObject\_GetAttrString(obj, "value"));

RETURN\_IF\_PYERROR();

RETURN\_NOT\_OK(internal::CIntFromPython(nanos.obj(), &value));

} else {

// Conversion to nanoseconds can overflow -> check multiply of microseconds

value = internal::PyDateTime\_to\_us(dt);

if (arrow::internal::MultiplyWithOverflow(value, 1000LL, &value)) {

return internal::InvalidValue(obj,

"out of bounds for nanosecond resolution");

}

// Adjust with offset and check for overflow

if (arrow::internal::SubtractWithOverflow(value, offset \* 1000000000LL,

&value)) {

return internal::InvalidValue(obj,

"out of bounds for nanosecond resolution");

}

}

break;

default:

return Status::UnknownError("Invalid time unit");

}

} else if (has\_numpy() && PyArray\_CheckAnyScalarExact(obj)) {

// validate that the numpy scalar has np.datetime64 dtype

ARROW\_ASSIGN\_OR\_RAISE(auto numpy\_type, NumPyScalarToArrowDataType(obj));

if (!numpy\_type->Equals(\*type)) {

return Status::NotImplemented("Expected np.datetime64 but got: ",

numpy\_type->ToString());

}

return reinterpret\_cast<PyDatetimeScalarObject\*>(obj)->obval;

} else {

RETURN\_NOT\_OK(internal::CIntFromPython(obj, &value));

}

return value;

}

static Result<MonthDayNanoIntervalType::MonthDayNanos> Convert(

const MonthDayNanoIntervalType\* /\*type\*/, const O& /\*options\*/, I obj) {

MonthDayNanoIntervalType::MonthDayNanos output;

bool found\_attrs = false;

RETURN\_NOT\_OK(PopulateMonthDayNano<MonthDayNanoField::kMonths>::Field(

obj, &output.months, &found\_attrs));

// on relativeoffset weeks is a property calculated from days. On

// DateOffset is a field on its own. timedelta doesn't have a weeks

// attribute.

PyObject\* pandas\_date\_offset\_type = internal::BorrowPandasDataOffsetType();

bool is\_date\_offset = pandas\_date\_offset\_type == (PyObject\*)Py\_TYPE(obj);

if (!is\_date\_offset) {

RETURN\_NOT\_OK(PopulateMonthDayNano<MonthDayNanoField::kDaysOnly>::Field(

obj, &output.days, &found\_attrs));

} else {

RETURN\_NOT\_OK(PopulateMonthDayNano<MonthDayNanoField::kWeeksAndDays>::Field(

obj, &output.days, &found\_attrs));

}

RETURN\_NOT\_OK(PopulateMonthDayNano<MonthDayNanoField::kNanoseconds>::Field(

obj, &output.nanoseconds, &found\_attrs));

// date\_offset can have zero fields.

if (found\_attrs || is\_date\_offset) {

return output;

}

if (PyTuple\_Check(obj) && PyTuple\_Size(obj) == 3) {

RETURN\_NOT\_OK(internal::CIntFromPython(PyTuple\_GET\_ITEM(obj, 0), &output.months,

"Months (tuple item #0) too large"));

RETURN\_NOT\_OK(internal::CIntFromPython(PyTuple\_GET\_ITEM(obj, 1), &output.days,

"Days (tuple item #1) too large"));

RETURN\_NOT\_OK(internal::CIntFromPython(PyTuple\_GET\_ITEM(obj, 2),

&output.nanoseconds,

"Nanoseconds (tuple item #2) too large"));

return output;

}

return Status::TypeError("No temporal attributes found on object.");

}

static Result<int64\_t> Convert(const DurationType\* type, const O&, I obj) {

int64\_t value;

if (PyDelta\_Check(obj)) {

auto dt = reinterpret\_cast<PyDateTime\_Delta\*>(obj);

switch (type->unit()) {

case TimeUnit::SECOND:

value = internal::PyDelta\_to\_s(dt);

break;

case TimeUnit::MILLI:

value = internal::PyDelta\_to\_ms(dt);

break;

case TimeUnit::MICRO: {

ARROW\_ASSIGN\_OR\_RAISE(value, internal::PyDelta\_to\_us(dt));

break;

}

case TimeUnit::NANO:

if (internal::IsPandasTimedelta(obj)) {

OwnedRef nanos(PyObject\_GetAttrString(obj, "value"));

RETURN\_IF\_PYERROR();

RETURN\_NOT\_OK(internal::CIntFromPython(nanos.obj(), &value));

} else {

ARROW\_ASSIGN\_OR\_RAISE(value, internal::PyDelta\_to\_ns(dt));

}

break;

default:

return Status::UnknownError("Invalid time unit");

}

} else if (has\_numpy() && PyArray\_CheckAnyScalarExact(obj)) {

// validate that the numpy scalar has np.datetime64 dtype

ARROW\_ASSIGN\_OR\_RAISE(auto numpy\_type, NumPyScalarToArrowDataType(obj));

if (!numpy\_type->Equals(\*type)) {

return Status::NotImplemented("Expected np.timedelta64 but got: ",

numpy\_type->ToString());

}

return reinterpret\_cast<PyTimedeltaScalarObject\*>(obj)->obval;

} else {

RETURN\_NOT\_OK(internal::CIntFromPython(obj, &value));

}

return value;

}

// The binary-like intermediate representation is PyBytesView because it keeps temporary

// python objects alive (non-contiguous memoryview) and stores whether the original

// object was unicode encoded or not, which is used for unicode -> bytes coercion if

// there is a non-unicode object observed.

static Status Convert(const BaseBinaryType\*, const O&, I obj, PyBytesView& view) {

return view.ParseString(obj);

}

static Status Convert(const BinaryViewType\*, const O&, I obj, PyBytesView& view) {

return view.ParseString(obj);

}

static Status Convert(const FixedSizeBinaryType\* type, const O&, I obj,

PyBytesView& view) {

ARROW\_RETURN\_NOT\_OK(view.ParseString(obj));

if (view.size != type->byte\_width()) {

std::stringstream ss;

ss << "expected to be length " << type->byte\_width() << " was " << view.size;

return internal::InvalidValue(obj, ss.str());

} else {

return Status::OK();

}

}

template <typename T>

static enable\_if\_t<is\_string\_type<T>::value || is\_string\_view\_type<T>::value, Status>

Convert(const T\*, const O& options, I obj, PyBytesView& view) {

if (options.strict) {

// Strict conversion, force output to be unicode / utf8 and validate that

// any binary values are utf8

ARROW\_RETURN\_NOT\_OK(view.ParseString(obj, true));

if (!view.is\_utf8) {

return internal::InvalidValue(obj, "was not a utf8 string");

}

return Status::OK();

} else {

// Non-strict conversion; keep track of whether values are unicode or bytes

return view.ParseString(obj);

}

}

static Result<bool> Convert(const DataType\* type, const O&, I obj) {

return Status::NotImplemented("PyValue::Convert is not implemented for type ", type);

}

};

// The base Converter class is a mixin with predefined behavior and constructors.

class PyConverter : public Converter<PyObject\*, PyConversionOptions> {

public:

// Iterate over the input values and defer the conversion to the Append method

Status Extend(PyObject\* values, int64\_t size, int64\_t offset = 0) override {

DCHECK\_GE(size, offset);

/// Ensure we've allocated enough space

RETURN\_NOT\_OK(this->Reserve(size - offset));

// Iterate over the items adding each one

return internal::VisitSequence(

values, offset,

[this](PyObject\* item, bool\* /\* unused \*/) { return this->Append(item); });

}

// Convert and append a sequence of values masked with a numpy array

Status ExtendMasked(PyObject\* values, PyObject\* mask, int64\_t size,

int64\_t offset = 0) override {

DCHECK\_GE(size, offset);

/// Ensure we've allocated enough space

RETURN\_NOT\_OK(this->Reserve(size - offset));

// Iterate over the items adding each one

return internal::VisitSequenceMasked(

values, mask, offset, [this](PyObject\* item, bool is\_masked, bool\* /\* unused \*/) {

if (is\_masked) {

return this->AppendNull();

} else {

// This will also apply the null-checking convention in the event

// that the value is not masked

return this->Append(item); // perhaps use AppendValue instead?

}

});

}

};

template <typename T, typename Enable = void>

class PyPrimitiveConverter;

template <typename T>

class PyListConverter;

template <typename U, typename Enable = void>

class PyDictionaryConverter;

class PyStructConverter;

template <typename T, typename Enable = void>

struct PyConverterTrait;

template <typename T>

struct PyConverterTrait<

T, enable\_if\_t<(!is\_nested\_type<T>::value && !is\_interval\_type<T>::value &&

!is\_extension\_type<T>::value) ||

std::is\_same<T, MonthDayNanoIntervalType>::value>> {

using type = PyPrimitiveConverter<T>;

};

template <typename T>

struct PyConverterTrait<

T, enable\_if\_t<is\_list\_like\_type<T>::value || is\_list\_view\_type<T>::value>> {

using type = PyListConverter<T>;

};

template <>

struct PyConverterTrait<StructType> {

using type = PyStructConverter;

};

template <>

struct PyConverterTrait<DictionaryType> {

template <typename T>

using dictionary\_type = PyDictionaryConverter<T>;

};

template <typename T>

class PyPrimitiveConverter<T, enable\_if\_null<T>>

: public PrimitiveConverter<T, PyConverter> {

public:

Status Append(PyObject\* value) override {

if (PyValue::IsNull(this->options\_, value)) {

return this->primitive\_builder\_->AppendNull();

} else if (arrow::py::is\_scalar(value)) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> scalar,

arrow::py::unwrap\_scalar(value));

if (scalar->is\_valid) {

return Status::Invalid("Cannot append scalar of type ", scalar->type->ToString(),

" to builder for type null");

} else {

return this->primitive\_builder\_->AppendNull();

}

} else {

ARROW\_ASSIGN\_OR\_RAISE(

auto converted, PyValue::Convert(this->primitive\_type\_, this->options\_, value));

return this->primitive\_builder\_->Append(converted);

}

}

};

template <typename T>

class PyPrimitiveConverter<

T, enable\_if\_t<is\_boolean\_type<T>::value || is\_number\_type<T>::value ||

is\_decimal\_type<T>::value || is\_date\_type<T>::value ||

is\_time\_type<T>::value ||

std::is\_same<MonthDayNanoIntervalType, T>::value>>

: public PrimitiveConverter<T, PyConverter> {

public:

Status Append(PyObject\* value) override {

// Since the required space has been already allocated in the Extend functions we can

// rely on the Unsafe builder API which improves the performance.

if (PyValue::IsNull(this->options\_, value)) {

this->primitive\_builder\_->UnsafeAppendNull();

} else if (arrow::py::is\_scalar(value)) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> scalar,

arrow::py::unwrap\_scalar(value));

ARROW\_RETURN\_NOT\_OK(this->primitive\_builder\_->AppendScalar(\*scalar));

} else {

ARROW\_ASSIGN\_OR\_RAISE(

auto converted, PyValue::Convert(this->primitive\_type\_, this->options\_, value));

this->primitive\_builder\_->UnsafeAppend(converted);

}

return Status::OK();

}

};

template <typename T>

class PyPrimitiveConverter<

T, enable\_if\_t<is\_timestamp\_type<T>::value || is\_duration\_type<T>::value>>

: public PrimitiveConverter<T, PyConverter> {

public:

Status Append(PyObject\* value) override {

if (PyValue::IsNull(this->options\_, value)) {

this->primitive\_builder\_->UnsafeAppendNull();

} else if (arrow::py::is\_scalar(value)) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> scalar,

arrow::py::unwrap\_scalar(value));

ARROW\_RETURN\_NOT\_OK(this->primitive\_builder\_->AppendScalar(\*scalar));

} else {

ARROW\_ASSIGN\_OR\_RAISE(

auto converted, PyValue::Convert(this->primitive\_type\_, this->options\_, value));

// Numpy NaT sentinels can be checked after the conversion

if (has\_numpy() && PyArray\_CheckAnyScalarExact(value) &&

PyValue::IsNaT(this->primitive\_type\_, converted)) {

this->primitive\_builder\_->UnsafeAppendNull();

} else {

this->primitive\_builder\_->UnsafeAppend(converted);

}

}

return Status::OK();

}

};

template <typename T>

class PyPrimitiveConverter<T, enable\_if\_t<std::is\_same<T, FixedSizeBinaryType>::value>>

: public PrimitiveConverter<T, PyConverter> {

public:

Status Append(PyObject\* value) override {

if (PyValue::IsNull(this->options\_, value)) {

this->primitive\_builder\_->UnsafeAppendNull();

} else if (arrow::py::is\_scalar(value)) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> scalar,

arrow::py::unwrap\_scalar(value));

ARROW\_RETURN\_NOT\_OK(this->primitive\_builder\_->AppendScalar(\*scalar));

} else {

ARROW\_RETURN\_NOT\_OK(

PyValue::Convert(this->primitive\_type\_, this->options\_, value, view\_));

ARROW\_RETURN\_NOT\_OK(this->primitive\_builder\_->ReserveData(view\_.size));

this->primitive\_builder\_->UnsafeAppend(view\_.bytes);

}

return Status::OK();

}

protected:

PyBytesView view\_;

};

template <typename T, typename Enable = void>

struct OffsetTypeTrait {

using type = typename T::offset\_type;

};

template <typename T>

struct OffsetTypeTrait<T, enable\_if\_binary\_view\_like<T>> {

using type = int64\_t;

};

template <typename T>

class PyPrimitiveConverter<

T, enable\_if\_t<is\_base\_binary\_type<T>::value || is\_binary\_view\_like\_type<T>::value>>

: public PrimitiveConverter<T, PyConverter> {

public:

using OffsetType = typename OffsetTypeTrait<T>::type;

Status Append(PyObject\* value) override {

if (PyValue::IsNull(this->options\_, value)) {

this->primitive\_builder\_->UnsafeAppendNull();

} else if (arrow::py::is\_scalar(value)) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> scalar,

arrow::py::unwrap\_scalar(value));

ARROW\_RETURN\_NOT\_OK(this->primitive\_builder\_->AppendScalar(\*scalar));

} else {

ARROW\_RETURN\_NOT\_OK(

PyValue::Convert(this->primitive\_type\_, this->options\_, value, view\_));

if (!view\_.is\_utf8) {

// observed binary value

observed\_binary\_ = true;

}

// Since we don't know the varying length input size in advance, we need to

// reserve space in the value builder one by one. ReserveData raises CapacityError

// if the value would not fit into the array.

ARROW\_RETURN\_NOT\_OK(this->primitive\_builder\_->ReserveData(view\_.size));

this->primitive\_builder\_->UnsafeAppend(view\_.bytes,

static\_cast<OffsetType>(view\_.size));

}

return Status::OK();

}

Result<std::shared\_ptr<Array>> ToArray() override {

ARROW\_ASSIGN\_OR\_RAISE(auto array, (PrimitiveConverter<T, PyConverter>::ToArray()));

if (observed\_binary\_) {

// if we saw any non-unicode, cast results to BinaryArray

auto binary\_type = TypeTraits<typename T::PhysicalType>::type\_singleton();

return array->View(binary\_type);

} else {

return array;

}

}

protected:

PyBytesView view\_;

bool observed\_binary\_ = false;

};

template <typename U>

class PyDictionaryConverter<U, enable\_if\_has\_c\_type<U>>

: public DictionaryConverter<U, PyConverter> {

public:

Status Append(PyObject\* value) override {

if (PyValue::IsNull(this->options\_, value)) {

return this->value\_builder\_->AppendNull();

} else if (arrow::py::is\_scalar(value)) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> scalar,

arrow::py::unwrap\_scalar(value));

return this->value\_builder\_->AppendScalar(\*scalar, 1);

} else {

ARROW\_ASSIGN\_OR\_RAISE(auto converted,

PyValue::Convert(this->value\_type\_, this->options\_, value));

return this->value\_builder\_->Append(converted);

}

}

};

template <typename U>

class PyDictionaryConverter<U, enable\_if\_has\_string\_view<U>>

: public DictionaryConverter<U, PyConverter> {

public:

Status Append(PyObject\* value) override {

if (PyValue::IsNull(this->options\_, value)) {

return this->value\_builder\_->AppendNull();

} else if (arrow::py::is\_scalar(value)) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> scalar,

arrow::py::unwrap\_scalar(value));

return this->value\_builder\_->AppendScalar(\*scalar, 1);

} else {

ARROW\_RETURN\_NOT\_OK(

PyValue::Convert(this->value\_type\_, this->options\_, value, view\_));

return this->value\_builder\_->Append(view\_.bytes, static\_cast<int32\_t>(view\_.size));

}

}

protected:

PyBytesView view\_;

};

template <typename T>

class PyListConverter : public ListConverter<T, PyConverter, PyConverterTrait> {

public:

Status Append(PyObject\* value) override {

if (PyValue::IsNull(this->options\_, value)) {

return this->list\_builder\_->AppendNull();

}

if (has\_numpy() && PyArray\_Check(value)) {

RETURN\_NOT\_OK(AppendNdarray(value));

} else if (PySequence\_Check(value)) {

RETURN\_NOT\_OK(AppendSequence(value));

} else if (PySet\_Check(value) || (Py\_TYPE(value) == &PyDictValues\_Type)) {

RETURN\_NOT\_OK(AppendIterable(value));

} else if (PyDict\_Check(value) && this->type()->id() == Type::MAP) {

// Branch to support Python Dict with `map` DataType.

auto items = PyDict\_Items(value);

OwnedRef item\_ref(items);

RETURN\_NOT\_OK(AppendSequence(items));

} else {

return internal::InvalidType(

value, "was not a sequence or recognized null for conversion to list type");

}

return ValidateBuilder(this->list\_type\_);

}

protected:

// MapType does not support args in the Append() method

Status AppendTo(const MapType\*, int64\_t size) { return this->list\_builder\_->Append(); }

// FixedSizeListType does not support args in the Append() method

Status AppendTo(const FixedSizeListType\*, int64\_t size) {

return this->list\_builder\_->Append();

}

// ListType requires the size argument in the Append() method

// in order to be convertible to a ListViewType. ListViewType

// requires the size argument in the Append() method always.

Status AppendTo(const BaseListType\*, int64\_t size) {

return this->list\_builder\_->Append(true, size);

}

Status ValidateBuilder(const MapType\*) {

if (this->list\_builder\_->key\_builder()->null\_count() > 0) {

return Status::Invalid("Invalid Map: key field cannot contain null values");

} else {

return Status::OK();

}

}

Status ValidateBuilder(const BaseListType\*) { return Status::OK(); }

Status AppendSequence(PyObject\* value) {

int64\_t size = static\_cast<int64\_t>(PySequence\_Size(value));

RETURN\_NOT\_OK(AppendTo(this->list\_type\_, size));

RETURN\_NOT\_OK(this->list\_builder\_->ValidateOverflow(size));

return this->value\_converter\_->Extend(value, size);

}

Status AppendIterable(PyObject\* value) {

auto size = static\_cast<int64\_t>(PyObject\_Size(value));

RETURN\_NOT\_OK(AppendTo(this->list\_type\_, size));

PyObject\* iterator = PyObject\_GetIter(value);

OwnedRef iter\_ref(iterator);

while (PyObject\* item = PyIter\_Next(iterator)) {

OwnedRef item\_ref(item);

RETURN\_NOT\_OK(this->value\_converter\_->Reserve(1));

RETURN\_NOT\_OK(this->value\_converter\_->Append(item));

}

return Status::OK();

}

Status AppendNdarray(PyObject\* value) {

PyArrayObject\* ndarray = reinterpret\_cast<PyArrayObject\*>(value);

if (PyArray\_NDIM(ndarray) != 1) {

return Status::Invalid("Can only convert 1-dimensional array values");

}

if (PyArray\_ISBYTESWAPPED(ndarray)) {

// TODO

return Status::NotImplemented("Byte-swapped arrays not supported");

}

const int64\_t size = PyArray\_SIZE(ndarray);

RETURN\_NOT\_OK(AppendTo(this->list\_type\_, size));

RETURN\_NOT\_OK(this->list\_builder\_->ValidateOverflow(size));

const auto value\_type = this->value\_converter\_->builder()->type();

switch (value\_type->id()) {

// If the value type does not match the expected NumPy dtype, then fall through

// to a slower PySequence-based path

#define LIST\_FAST\_CASE(TYPE\_ID, TYPE, NUMPY\_TYPE) \

case Type::TYPE\_ID: { \

if (PyArray\_DESCR(ndarray)->type\_num != NUMPY\_TYPE) { \

return this->value\_converter\_->Extend(value, size); \

} \

return AppendNdarrayTyped<TYPE, NUMPY\_TYPE>(ndarray); \

}

LIST\_FAST\_CASE(BOOL, BooleanType, NPY\_BOOL)

LIST\_FAST\_CASE(UINT8, UInt8Type, NPY\_UINT8)

LIST\_FAST\_CASE(INT8, Int8Type, NPY\_INT8)

LIST\_FAST\_CASE(UINT16, UInt16Type, NPY\_UINT16)

LIST\_FAST\_CASE(INT16, Int16Type, NPY\_INT16)

LIST\_FAST\_CASE(UINT32, UInt32Type, NPY\_UINT32)

LIST\_FAST\_CASE(INT32, Int32Type, NPY\_INT32)

LIST\_FAST\_CASE(UINT64, UInt64Type, NPY\_UINT64)

LIST\_FAST\_CASE(INT64, Int64Type, NPY\_INT64)

LIST\_FAST\_CASE(HALF\_FLOAT, HalfFloatType, NPY\_FLOAT16)

LIST\_FAST\_CASE(FLOAT, FloatType, NPY\_FLOAT)

LIST\_FAST\_CASE(DOUBLE, DoubleType, NPY\_DOUBLE)

LIST\_FAST\_CASE(TIMESTAMP, TimestampType, NPY\_DATETIME)

LIST\_FAST\_CASE(DURATION, DurationType, NPY\_TIMEDELTA)

#undef LIST\_FAST\_CASE

default: {

return this->value\_converter\_->Extend(value, size);

}

}

}

template <typename ArrowType, int NUMPY\_TYPE>

Status AppendNdarrayTyped(PyArrayObject\* ndarray) {

// no need to go through the conversion

using NumpyTrait = internal::npy\_traits<NUMPY\_TYPE>;

using NumpyType = typename NumpyTrait::value\_type;

using ValueBuilderType = typename TypeTraits<ArrowType>::BuilderType;

const bool null\_sentinels\_possible =

// Always treat Numpy's NaT as null

NUMPY\_TYPE == NPY\_DATETIME || NUMPY\_TYPE == NPY\_TIMEDELTA ||

// Observing pandas's null sentinels

(this->options\_.from\_pandas && NumpyTrait::supports\_nulls);

auto value\_builder =

checked\_cast<ValueBuilderType\*>(this->value\_converter\_->builder().get());

Ndarray1DIndexer<NumpyType> values(ndarray);

if (null\_sentinels\_possible) {

for (int64\_t i = 0; i < values.size(); ++i) {

if (NumpyTrait::isnull(values[i])) {

RETURN\_NOT\_OK(value\_builder->AppendNull());

} else {

RETURN\_NOT\_OK(value\_builder->Append(values[i]));

}

}

} else if (!values.is\_strided()) {

RETURN\_NOT\_OK(value\_builder->AppendValues(values.data(), values.size()));

} else {

for (int64\_t i = 0; i < values.size(); ++i) {

RETURN\_NOT\_OK(value\_builder->Append(values[i]));

}

}

return Status::OK();

}

};

class PyStructConverter : public StructConverter<PyConverter, PyConverterTrait> {

public:

Status Append(PyObject\* value) override {

if (PyValue::IsNull(this->options\_, value)) {

return this->struct\_builder\_->AppendNull();

} else if (arrow::py::is\_scalar(value)) {

ARROW\_ASSIGN\_OR\_RAISE(std::shared\_ptr<Scalar> scalar,

arrow::py::unwrap\_scalar(value));

return this->struct\_builder\_->AppendScalar(\*scalar);

}

switch (input\_kind\_) {

case InputKind::DICT:

RETURN\_NOT\_OK(AppendDict(value));

return this->struct\_builder\_->Append();

case InputKind::TUPLE:

RETURN\_NOT\_OK(AppendTuple(value));

return this->struct\_builder\_->Append();

case InputKind::ITEMS:

RETURN\_NOT\_OK(AppendItems(value));

return this->struct\_builder\_->Append();

default:

RETURN\_NOT\_OK(InferInputKind(value));

return Append(value);

}

}

protected:

Status Init(MemoryPool\* pool) override {

RETURN\_NOT\_OK((StructConverter<PyConverter, PyConverterTrait>::Init(pool)));

// This implementation will check the child values before appending itself,

// so no rewind is necessary

this->rewind\_on\_overflow\_ = false;

// Store the field names as a PyObjects for dict matching

num\_fields\_ = this->struct\_type\_->num\_fields();

bytes\_field\_names\_.reset(PyList\_New(num\_fields\_));

unicode\_field\_names\_.reset(PyList\_New(num\_fields\_));

RETURN\_IF\_PYERROR();

for (int i = 0; i < num\_fields\_; i++) {

const auto& field\_name = this->struct\_type\_->field(i)->name();

PyObject\* bytes = PyBytes\_FromStringAndSize(field\_name.c\_str(), field\_name.size());

PyObject\* unicode =

PyUnicode\_FromStringAndSize(field\_name.c\_str(), field\_name.size());

RETURN\_IF\_PYERROR();

PyList\_SET\_ITEM(bytes\_field\_names\_.obj(), i, bytes);

PyList\_SET\_ITEM(unicode\_field\_names\_.obj(), i, unicode);

}

return Status::OK();

}

Status InferInputKind(PyObject\* value) {

// Infer input object's type, note that heterogeneous sequences are not allowed

if (PyDict\_Check(value)) {

input\_kind\_ = InputKind::DICT;

} else if (PyTuple\_Check(value)) {

input\_kind\_ = InputKind::TUPLE;

} else if (PySequence\_Check(value)) {

input\_kind\_ = InputKind::ITEMS;

} else {

return internal::InvalidType(value,

"was not a dict, tuple, or recognized null value "

"for conversion to struct type");

}

return Status::OK();

}

Status InferKeyKind(PyObject\* items) {

for (int i = 0; i < PySequence\_Length(items); i++) {

// retrieve the key from the passed key-value pairs

ARROW\_ASSIGN\_OR\_RAISE(auto pair, GetKeyValuePair(items, i));

// check key exists between the unicode field names

bool do\_contain = PySequence\_Contains(unicode\_field\_names\_.obj(), pair.first);

RETURN\_IF\_PYERROR();

if (do\_contain) {

key\_kind\_ = KeyKind::UNICODE;

return Status::OK();

}

// check key exists between the bytes field names

do\_contain = PySequence\_Contains(bytes\_field\_names\_.obj(), pair.first);

RETURN\_IF\_PYERROR();

if (do\_contain) {

key\_kind\_ = KeyKind::BYTES;

return Status::OK();

}

}

return Status::OK();

}

Status AppendEmpty() {

for (int i = 0; i < num\_fields\_; i++) {

RETURN\_NOT\_OK(this->children\_[i]->Append(Py\_None));

}

return Status::OK();

}

Status AppendTuple(PyObject\* tuple) {

if (!PyTuple\_Check(tuple)) {

return internal::InvalidType(tuple, "was expecting a tuple");

}

if (PyTuple\_GET\_SIZE(tuple) != num\_fields\_) {

return Status::Invalid("Tuple size must be equal to number of struct fields");

}

for (int i = 0; i < num\_fields\_; i++) {

PyObject\* value = PyTuple\_GET\_ITEM(tuple, i);

RETURN\_NOT\_OK(this->children\_[i]->Append(value));

}

return Status::OK();

}

Status AppendDict(PyObject\* dict) {

if (!PyDict\_Check(dict)) {

return internal::InvalidType(dict, "was expecting a dict");

}

switch (key\_kind\_) {

case KeyKind::UNICODE:

return AppendDict(dict, unicode\_field\_names\_.obj());

case KeyKind::BYTES:

return AppendDict(dict, bytes\_field\_names\_.obj());

default:

OwnedRef item\_ref(PyDict\_Items(dict));

RETURN\_NOT\_OK(InferKeyKind(item\_ref.obj()));

if (key\_kind\_ == KeyKind::UNKNOWN) {

// was unable to infer the type which means that all keys are absent

return AppendEmpty();

} else {

return AppendDict(dict);

}

}

}

Status AppendItems(PyObject\* items) {

if (!PySequence\_Check(items)) {

return internal::InvalidType(items, "was expecting a sequence of key-value items");

}

switch (key\_kind\_) {

case KeyKind::UNICODE:

return AppendItems(items, unicode\_field\_names\_.obj());

case KeyKind::BYTES:

return AppendItems(items, bytes\_field\_names\_.obj());

default:

RETURN\_NOT\_OK(InferKeyKind(items));

if (key\_kind\_ == KeyKind::UNKNOWN) {

// was unable to infer the type which means that all keys are absent

return AppendEmpty();

} else {

return AppendItems(items);

}

}

}

Status AppendDict(PyObject\* dict, PyObject\* field\_names) {

// NOTE we're ignoring any extraneous dict items

for (int i = 0; i < num\_fields\_; i++) {

PyObject\* name = PyList\_GetItemRef(field\_names, i);

RETURN\_IF\_PYERROR();

OwnedRef nameref(name);

PyObject\* value;

PyDict\_GetItemRef(dict, name, &value);

RETURN\_IF\_PYERROR();

OwnedRef valueref(value);

RETURN\_NOT\_OK(this->children\_[i]->Append(value ? value : Py\_None));

}

return Status::OK();

}

Result<std::pair<PyObject\*, PyObject\*>> GetKeyValuePair(PyObject\* seq, int index) {

PyObject\* pair = PySequence\_GetItem(seq, index);

RETURN\_IF\_PYERROR();

OwnedRef pair\_ref(pair); // ensure reference count is decreased at scope end

if (!PyTuple\_Check(pair) || PyTuple\_Size(pair) != 2) {

return internal::InvalidType(pair, "was expecting tuple of (key, value) pair");

}

PyObject\* key = PyTuple\_GetItem(pair, 0);

RETURN\_IF\_PYERROR();

PyObject\* value = PyTuple\_GetItem(pair, 1);

RETURN\_IF\_PYERROR();

return std::make\_pair(key, value);

}

Status AppendItems(PyObject\* items, PyObject\* field\_names) {

auto length = static\_cast<int>(PySequence\_Size(items));

RETURN\_IF\_PYERROR();

// append the values for the defined fields

for (int i = 0; i < std::min(num\_fields\_, length); i++) {

// retrieve the key-value pair

ARROW\_ASSIGN\_OR\_RAISE(auto pair, GetKeyValuePair(items, i));

// validate that the key and the field name are equal

PyObject\* name = PyList\_GetItemRef(field\_names, i);

RETURN\_IF\_PYERROR();

OwnedRef nameref(name);

bool are\_equal = PyObject\_RichCompareBool(pair.first, name, Py\_EQ);

RETURN\_IF\_PYERROR();

// finally append to the respective child builder

if (are\_equal) {

RETURN\_NOT\_OK(this->children\_[i]->Append(pair.second));

} else {

ARROW\_ASSIGN\_OR\_RAISE(auto key\_view, PyBytesView::FromString(pair.first));

ARROW\_ASSIGN\_OR\_RAISE(auto name\_view, PyBytesView::FromString(name));

return Status::Invalid("The expected field name is `", name\_view.bytes, "` but `",

key\_view.bytes, "` was given");

}

}

// insert null values for missing fields

for (int i = length; i < num\_fields\_; i++) {

RETURN\_NOT\_OK(this->children\_[i]->AppendNull());

}

return Status::OK();

}

// Whether we're converting from a sequence of dicts or tuples or list of pairs

enum class InputKind { UNKNOWN, DICT, TUPLE, ITEMS } input\_kind\_ = InputKind::UNKNOWN;

// Whether the input dictionary keys' type is python bytes or unicode

enum class KeyKind { UNKNOWN, BYTES, UNICODE } key\_kind\_ = KeyKind::UNKNOWN;

// Store the field names as a PyObjects for dict matching

OwnedRef bytes\_field\_names\_;

OwnedRef unicode\_field\_names\_;

// Store the number of fields for later reuse

int num\_fields\_;

};

// Convert \*obj\* to a sequence if necessary

// Fill \*size\* to its length. If >= 0 on entry, \*size\* is an upper size

// bound that may lead to truncation.

Status ConvertToSequenceAndInferSize(PyObject\* obj, PyObject\*\* seq, int64\_t\* size) {

if (PySequence\_Check(obj)) {

// obj is already a sequence

int64\_t real\_size = static\_cast<int64\_t>(PySequence\_Size(obj));

RETURN\_IF\_PYERROR();

if (\*size < 0) {

\*size = real\_size;

} else {

\*size = std::min(real\_size, \*size);

}

Py\_INCREF(obj);

\*seq = obj;

} else if (\*size < 0) {

// unknown size, exhaust iterator

\*seq = PySequence\_List(obj);

RETURN\_IF\_PYERROR();

\*size = static\_cast<int64\_t>(PyList\_GET\_SIZE(\*seq));

} else {

// size is known but iterator could be infinite

Py\_ssize\_t i, n = \*size;

PyObject\* iter = PyObject\_GetIter(obj);

RETURN\_IF\_PYERROR();

OwnedRef iter\_ref(iter);

PyObject\* lst = PyList\_New(n);

RETURN\_IF\_PYERROR();

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

PyObject\* item = PyIter\_Next(iter);

if (!item) {

// either an error occurred or the iterator ended

RETURN\_IF\_PYERROR();

break;

}

PyList\_SET\_ITEM(lst, i, item);

}

// Shrink list if len(iterator) < size

if (i < n && PyList\_SetSlice(lst, i, n, NULL)) {

Py\_DECREF(lst);

RETURN\_IF\_PYERROR();

}

\*seq = lst;

\*size = std::min<int64\_t>(i, \*size);

}

return Status::OK();

}

} // namespace

Result<std::shared\_ptr<ChunkedArray>> ConvertPySequence(PyObject\* obj, PyObject\* mask,

PyConversionOptions options,

MemoryPool\* pool) {

PyAcquireGIL lock;

PyObject\* seq = nullptr;

OwnedRef tmp\_seq\_nanny;

ARROW\_ASSIGN\_OR\_RAISE(auto is\_pandas\_imported, internal::IsModuleImported("pandas"));

if (is\_pandas\_imported) {

// If pandas has been already imported initialize the static pandas objects to

// support converting from pd.Timedelta and pd.Timestamp objects

internal::InitPandasStaticData();

}

int64\_t size = options.size;

RETURN\_NOT\_OK(ConvertToSequenceAndInferSize(obj, &seq, &size));

tmp\_seq\_nanny.reset(seq);

// In some cases, type inference may be "loose", like strings. If the user

// passed pa.string(), then we will error if we encounter any non-UTF8

// value. If not, then we will allow the result to be a BinaryArray

if (options.type == nullptr) {

ARROW\_ASSIGN\_OR\_RAISE(options.type, InferArrowType(seq, mask, options.from\_pandas));

options.strict = false;

} else {

options.strict = true;

}

DCHECK\_GE(size, 0);

ARROW\_ASSIGN\_OR\_RAISE(auto converter, (MakeConverter<PyConverter, PyConverterTrait>(

options.type, options, pool)));

if (converter->may\_overflow()) {

// The converter hierarchy contains binary- or list-like builders which can overflow

// depending on the input values. Wrap the converter with a chunker which detects

// the overflow and automatically creates new chunks.

ARROW\_ASSIGN\_OR\_RAISE(auto chunked\_converter, MakeChunker(std::move(converter)));

if (mask != nullptr && mask != Py\_None) {

RETURN\_NOT\_OK(chunked\_converter->ExtendMasked(seq, mask, size));

} else {

RETURN\_NOT\_OK(chunked\_converter->Extend(seq, size));

}

return chunked\_converter->ToChunkedArray();

} else {

// If the converter can't overflow spare the capacity error checking on the hot-path,

// this improves the performance roughly by ~10% for primitive types.

if (mask != nullptr && mask != Py\_None) {

RETURN\_NOT\_OK(converter->ExtendMasked(seq, mask, size));

} else {

RETURN\_NOT\_OK(converter->Extend(seq, size));

}

return converter->ToChunkedArray();

}

}

} // namespace py

} // namespace arrow