// 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 <memory>

#include <optional>

#include <sstream>

#include <string>

#include "platform.h"

#include "arrow/array.h"

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

#include "arrow/table.h"

#include "arrow/util/decimal.h"

#include "arrow/util/logging.h"

#include "arrow/python/arrow\_to\_pandas.h"

#include "arrow/python/decimal.h"

#include "arrow/python/helpers.h"

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

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

#include "arrow/python/python\_test.h"

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

#define ASSERT\_EQ(x, y) \

{ \

auto&& \_left = (x); \

auto&& \_right = (y); \

if (\_left != \_right) { \

return Status::Invalid("Expected equality between `", #x, "` and `", #y, \

"`, but ", arrow::py::testing::ToString(\_left), \

" != ", arrow::py::testing::ToString(\_right)); \

} \

}

#define ASSERT\_NE(x, y) \

{ \

auto&& \_left = (x); \

auto&& \_right = (y); \

if (\_left == \_right) { \

return Status::Invalid("Expected inequality between `", #x, "` and `", #y, \

"`, but ", arrow::py::testing::ToString(\_left), \

" == ", arrow::py::testing::ToString(\_right)); \

} \

}

#define ASSERT\_FALSE(v) \

{ \

auto&& \_v = (v); \

if (!!\_v) { \

return Status::Invalid("Expected `", #v, "` to evaluate to false, but got ", \

arrow::py::testing::ToString(\_v)); \

} \

}

#define ASSERT\_TRUE(v) \

{ \

auto&& \_v = (v); \

if (!\_v) { \

return Status::Invalid("Expected `", #v, "` to evaluate to true, but got ", \

arrow::py::testing::ToString(\_v)); \

} \

}

#define ASSERT\_FALSE\_MSG(v, msg) \

{ \

auto&& \_v = (v); \

if (!!\_v) { \

return Status::Invalid("Expected `", #v, "` to evaluate to false, but got ", \

arrow::py::testing::ToString(\_v), ": ", msg); \

} \

}

#define ASSERT\_TRUE\_MSG(v, msg) \

{ \

auto&& \_v = (v); \

if (!\_v) { \

return Status::Invalid("Expected `", #v, "` to evaluate to true, but got ", \

arrow::py::testing::ToString(\_v), ": ", msg); \

} \

}

#define ASSERT\_OK(expr) \

{ \

for (::arrow::Status \_st = ::arrow::internal::GenericToStatus((expr)); !\_st.ok();) \

return Status::Invalid("`", #expr, "` failed with ", \_st.ToString()); \

}

#define ASSERT\_RAISES(code, expr) \

{ \

for (::arrow::Status \_st\_expr = ::arrow::internal::GenericToStatus((expr)); \

!\_st\_expr.Is##code();) \

return Status::Invalid("Expected `", #expr, "` to fail with ", #code, \

", but got ", \_st\_expr.ToString()); \

}

namespace arrow {

using internal::checked\_cast;

namespace py {

namespace testing {

// ARROW-17938: Some standard libraries have ambiguous operator<<(nullptr\_t),

// work around it using a custom printer function.

template <typename T>

std::string ToString(const T& t) {

std::stringstream ss;

ss << t;

return ss.str();

}

template <>

std::string ToString(const std::nullptr\_t&) {

return "nullptr";

}

namespace {

Status TestOwnedRefMoves() {

std::vector<OwnedRef> vec;

PyObject \*u, \*v;

u = PyList\_New(0);

v = PyList\_New(0);

{

OwnedRef ref(u);

vec.push\_back(std::move(ref));

ASSERT\_EQ(ref.obj(), nullptr);

}

vec.emplace\_back(v);

ASSERT\_EQ(Py\_REFCNT(u), 1);

ASSERT\_EQ(Py\_REFCNT(v), 1);

return Status::OK();

}

Status TestOwnedRefNoGILMoves() {

PyAcquireGIL lock;

lock.release();

{

std::vector<OwnedRef> vec;

PyObject \*u, \*v;

{

lock.acquire();

u = PyList\_New(0);

v = PyList\_New(0);

lock.release();

}

{

OwnedRefNoGIL ref(u);

vec.push\_back(std::move(ref));

ASSERT\_EQ(ref.obj(), nullptr);

}

vec.emplace\_back(v);

ASSERT\_EQ(Py\_REFCNT(u), 1);

ASSERT\_EQ(Py\_REFCNT(v), 1);

return Status::OK();

}

}

std::string FormatPythonException(const std::string& exc\_class\_name,

const std::string& exc\_value) {

std::stringstream ss;

ss << "Python exception: ";

ss << exc\_class\_name;

ss << ": ";

ss << exc\_value;

ss << "\n";

return ss.str();

}

Status TestCheckPyErrorStatus() {

Status st;

std::string expected\_detail = "";

auto check\_error = [](Status& st, const char\* expected\_message = "some error",

std::string expected\_detail = "") {

st = CheckPyError();

ASSERT\_EQ(st.message(), expected\_message);

ASSERT\_FALSE(PyErr\_Occurred());

if (expected\_detail.size() > 0) {

auto detail = st.detail();

ASSERT\_NE(detail, nullptr);

ASSERT\_EQ(detail->ToString(), expected\_detail);

}

return Status::OK();

};

for (PyObject\* exc\_type : {PyExc\_Exception, PyExc\_SyntaxError}) {

PyErr\_SetString(exc\_type, "some error");

ASSERT\_OK(check\_error(st));

ASSERT\_TRUE(st.IsUnknownError());

}

PyErr\_SetString(PyExc\_TypeError, "some error");

ASSERT\_OK(

check\_error(st, "some error", FormatPythonException("TypeError", "some error")));

ASSERT\_TRUE(st.IsTypeError());

PyErr\_SetString(PyExc\_ValueError, "some error");

ASSERT\_OK(check\_error(st));

ASSERT\_TRUE(st.IsInvalid());

PyErr\_SetString(PyExc\_KeyError, "some error");

ASSERT\_OK(check\_error(st, "'some error'"));

ASSERT\_TRUE(st.IsKeyError());

for (PyObject\* exc\_type : {PyExc\_OSError, PyExc\_IOError}) {

PyErr\_SetString(exc\_type, "some error");

ASSERT\_OK(check\_error(st));

ASSERT\_TRUE(st.IsIOError());

}

PyErr\_SetString(PyExc\_NotImplementedError, "some error");

ASSERT\_OK(check\_error(st, "some error",

FormatPythonException("NotImplementedError", "some error")));

ASSERT\_TRUE(st.IsNotImplemented());

// No override if a specific status code is given

PyErr\_SetString(PyExc\_TypeError, "some error");

st = CheckPyError(StatusCode::SerializationError);

ASSERT\_TRUE(st.IsSerializationError());

ASSERT\_EQ(st.message(), "some error");

ASSERT\_FALSE(PyErr\_Occurred());

return Status::OK();

}

Status TestCheckPyErrorStatusNoGIL() {

PyAcquireGIL lock;

{

Status st;

PyErr\_SetString(PyExc\_ZeroDivisionError, "zzzt");

st = ConvertPyError();

ASSERT\_FALSE(PyErr\_Occurred());

lock.release();

ASSERT\_TRUE(st.IsUnknownError());

ASSERT\_EQ(st.message(), "zzzt");

ASSERT\_EQ(st.detail()->ToString(),

FormatPythonException("ZeroDivisionError", "zzzt"));

return Status::OK();

}

}

Status TestRestorePyErrorBasics() {

PyErr\_SetString(PyExc\_ZeroDivisionError, "zzzt");

auto st = ConvertPyError();

ASSERT\_FALSE(PyErr\_Occurred());

ASSERT\_TRUE(st.IsUnknownError());

ASSERT\_EQ(st.message(), "zzzt");

ASSERT\_EQ(st.detail()->ToString(), FormatPythonException("ZeroDivisionError", "zzzt"));

RestorePyError(st);

ASSERT\_TRUE(PyErr\_Occurred());

PyObject\* exc\_type;

PyObject\* exc\_value;

PyObject\* exc\_traceback;

PyErr\_Fetch(&exc\_type, &exc\_value, &exc\_traceback);

ASSERT\_TRUE(PyErr\_GivenExceptionMatches(exc\_type, PyExc\_ZeroDivisionError));

std::string py\_message;

ASSERT\_OK(internal::PyObject\_StdStringStr(exc\_value, &py\_message));

ASSERT\_EQ(py\_message, "zzzt");

return Status::OK();

}

Status TestPyBufferInvalidInputObject() {

std::shared\_ptr<Buffer> res;

PyObject\* input = Py\_None;

auto old\_refcnt = Py\_REFCNT(input);

{

Status st = PyBuffer::FromPyObject(input).status();

ASSERT\_TRUE\_MSG(IsPyError(st), st.ToString());

ASSERT\_FALSE(PyErr\_Occurred());

}

ASSERT\_EQ(old\_refcnt, Py\_REFCNT(input));

return Status::OK();

}

// Because of how it is declared, the Numpy C API instance initialized

// within libarrow\_python.dll may not be visible in this test under Windows

// ("unresolved external symbol arrow\_ARRAY\_API referenced").

#ifndef \_WIN32

Status TestPyBufferNumpyArray() {

npy\_intp dims[1] = {10};

OwnedRef arr\_ref(PyArray\_SimpleNew(1, dims, NPY\_FLOAT));

PyObject\* arr = arr\_ref.obj();

ASSERT\_NE(arr, nullptr);

auto old\_refcnt = Py\_REFCNT(arr);

auto buf = std::move(PyBuffer::FromPyObject(arr)).ValueOrDie();

ASSERT\_TRUE(buf->is\_cpu());

ASSERT\_EQ(buf->data(), PyArray\_DATA(reinterpret\_cast<PyArrayObject\*>(arr)));

ASSERT\_TRUE(buf->is\_mutable());

ASSERT\_EQ(buf->mutable\_data(), buf->data());

ASSERT\_EQ(old\_refcnt + 1, Py\_REFCNT(arr));

buf.reset();

ASSERT\_EQ(old\_refcnt, Py\_REFCNT(arr));

// Read-only

PyArray\_CLEARFLAGS(reinterpret\_cast<PyArrayObject\*>(arr), NPY\_ARRAY\_WRITEABLE);

buf = std::move(PyBuffer::FromPyObject(arr)).ValueOrDie();

ASSERT\_TRUE(buf->is\_cpu());

ASSERT\_EQ(buf->data(), PyArray\_DATA(reinterpret\_cast<PyArrayObject\*>(arr)));

ASSERT\_FALSE(buf->is\_mutable());

ASSERT\_EQ(old\_refcnt + 1, Py\_REFCNT(arr));

buf.reset();

ASSERT\_EQ(old\_refcnt, Py\_REFCNT(arr));

return Status::OK();

}

Status TestNumPyBufferNumpyArray() {

npy\_intp dims[1] = {10};

OwnedRef arr\_ref(PyArray\_SimpleNew(1, dims, NPY\_FLOAT));

PyObject\* arr = arr\_ref.obj();

ASSERT\_NE(arr, nullptr);

auto old\_refcnt = Py\_REFCNT(arr);

auto buf = std::make\_shared<NumPyBuffer>(arr);

ASSERT\_TRUE(buf->is\_cpu());

ASSERT\_EQ(buf->data(), PyArray\_DATA(reinterpret\_cast<PyArrayObject\*>(arr)));

ASSERT\_TRUE(buf->is\_mutable());

ASSERT\_EQ(buf->mutable\_data(), buf->data());

ASSERT\_EQ(old\_refcnt + 1, Py\_REFCNT(arr));

buf.reset();

ASSERT\_EQ(old\_refcnt, Py\_REFCNT(arr));

// Read-only

PyArray\_CLEARFLAGS(reinterpret\_cast<PyArrayObject\*>(arr), NPY\_ARRAY\_WRITEABLE);

buf = std::make\_shared<NumPyBuffer>(arr);

ASSERT\_TRUE(buf->is\_cpu());

ASSERT\_EQ(buf->data(), PyArray\_DATA(reinterpret\_cast<PyArrayObject\*>(arr)));

ASSERT\_FALSE(buf->is\_mutable());

ASSERT\_EQ(old\_refcnt + 1, Py\_REFCNT(arr));

buf.reset();

ASSERT\_EQ(old\_refcnt, Py\_REFCNT(arr));

return Status::OK();

}

#endif

Status TestPythonDecimalToString() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("-39402950693754869342983");

PyObject\* python\_object =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

ASSERT\_NE(python\_object, nullptr);

std::string string\_result;

ASSERT\_OK(internal::PythonDecimalToString(python\_object, &string\_result));

return Status::OK();

}

Status TestInferPrecisionAndScale() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("-394029506937548693.42983");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

internal::DecimalMetadata metadata;

ASSERT\_OK(metadata.Update(python\_decimal));

const auto expected\_precision =

static\_cast<int32\_t>(decimal\_string.size() - 2); // 1 for -, 1 for .

const int32\_t expected\_scale = 5;

ASSERT\_EQ(expected\_precision, metadata.precision());

ASSERT\_EQ(expected\_scale, metadata.scale());

return Status::OK();

}

Status TestInferPrecisionAndNegativeScale() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("-3.94042983E+10");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

internal::DecimalMetadata metadata;

ASSERT\_OK(metadata.Update(python\_decimal));

const auto expected\_precision = 11;

const int32\_t expected\_scale = 0;

ASSERT\_EQ(expected\_precision, metadata.precision());

ASSERT\_EQ(expected\_scale, metadata.scale());

return Status::OK();

}

Status TestInferAllLeadingZeros() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("0.001");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

internal::DecimalMetadata metadata;

ASSERT\_OK(metadata.Update(python\_decimal));

ASSERT\_EQ(3, metadata.precision());

ASSERT\_EQ(3, metadata.scale());

return Status::OK();

}

Status TestInferAllLeadingZerosExponentialNotationPositive() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("0.01E5");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

internal::DecimalMetadata metadata;

ASSERT\_OK(metadata.Update(python\_decimal));

ASSERT\_EQ(4, metadata.precision());

ASSERT\_EQ(0, metadata.scale());

return Status::OK();

}

Status TestInferAllLeadingZerosExponentialNotationNegative() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("0.01E3");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

internal::DecimalMetadata metadata;

ASSERT\_OK(metadata.Update(python\_decimal));

ASSERT\_EQ(2, metadata.precision());

ASSERT\_EQ(0, metadata.scale());

return Status::OK();

}

Status TestObjectBlockWriteFails() {

StringBuilder builder;

const char value[] = {'\xf1', '\0'};

for (int i = 0; i < 1000; ++i) {

ASSERT\_OK(builder.Append(value, static\_cast<int32\_t>(strlen(value))));

}

std::shared\_ptr<Array> arr;

ASSERT\_OK(builder.Finish(&arr));

auto f1 = field("f1", utf8());

auto f2 = field("f2", utf8());

auto f3 = field("f3", utf8());

std::vector<std::shared\_ptr<Field>> fields = {f1, f2, f3};

std::vector<std::shared\_ptr<Array>> cols = {arr, arr, arr};

auto schema = ::arrow::schema(fields);

auto table = Table::Make(schema, cols);

Status st;

Py\_BEGIN\_ALLOW\_THREADS;

PyObject\* out;

PandasOptions options;

options.use\_threads = true;

st = ConvertTableToPandas(options, table, &out);

Py\_END\_ALLOW\_THREADS;

ASSERT\_RAISES(UnknownError, st);

return Status::OK();

}

Status TestMixedTypeFails() {

OwnedRef list\_ref(PyList\_New(3));

PyObject\* list = list\_ref.obj();

ASSERT\_NE(list, nullptr);

PyObject\* str = PyUnicode\_FromString("abc");

ASSERT\_NE(str, nullptr);

PyObject\* integer = PyLong\_FromLong(1234L);

ASSERT\_NE(integer, nullptr);

PyObject\* doub = PyFloat\_FromDouble(123.0234);

ASSERT\_NE(doub, nullptr);

// This steals a reference to each object, so we don't need to decref them later

// just the list

ASSERT\_EQ(PyList\_SetItem(list, 0, str), 0);

ASSERT\_EQ(PyList\_SetItem(list, 1, integer), 0);

ASSERT\_EQ(PyList\_SetItem(list, 2, doub), 0);

ASSERT\_RAISES(TypeError, ConvertPySequence(list, nullptr, {}));

return Status::OK();

}

template <typename DecimalValue>

Status DecimalTestFromPythonDecimalRescale(std::shared\_ptr<DataType> type,

PyObject\* python\_decimal,

std::optional<int> expected) {

DecimalValue value;

const auto& decimal\_type = checked\_cast<const DecimalType&>(\*type);

if (expected.has\_value()) {

ASSERT\_OK(internal::DecimalFromPythonDecimal(python\_decimal, decimal\_type, &value));

ASSERT\_EQ(expected.value(), value);

ASSERT\_OK(internal::DecimalFromPyObject(python\_decimal, decimal\_type, &value));

ASSERT\_EQ(expected.value(), value);

} else {

ASSERT\_RAISES(Invalid, internal::DecimalFromPythonDecimal(python\_decimal,

decimal\_type, &value));

ASSERT\_RAISES(Invalid,

internal::DecimalFromPyObject(python\_decimal, decimal\_type, &value));

}

return Status::OK();

}

Status TestFromPythonDecimalRescaleNotTruncateable() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("1.001");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

// We fail when truncating values that would lose data if cast to a decimal type with

// lower scale

ASSERT\_OK(DecimalTestFromPythonDecimalRescale<Decimal128>(::arrow::decimal128(10, 2),

python\_decimal, {}));

ASSERT\_OK(DecimalTestFromPythonDecimalRescale<Decimal256>(::arrow::decimal256(10, 2),

python\_decimal, {}));

return Status::OK();

}

Status TestFromPythonDecimalRescaleTruncateable() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("1.000");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

// We allow truncation of values that do not lose precision when dividing by 10 \* the

// difference between the scales, e.g., 1.000 -> 1.00

ASSERT\_OK(DecimalTestFromPythonDecimalRescale<Decimal128>(::arrow::decimal128(10, 2),

python\_decimal, 100));

ASSERT\_OK(DecimalTestFromPythonDecimalRescale<Decimal256>(::arrow::decimal256(10, 2),

python\_decimal, 100));

return Status::OK();

}

Status TestFromPythonNegativeDecimalRescale() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("-1.000");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

ASSERT\_OK(DecimalTestFromPythonDecimalRescale<Decimal128>(::arrow::decimal128(10, 9),

python\_decimal, -1000000000));

ASSERT\_OK(DecimalTestFromPythonDecimalRescale<Decimal256>(::arrow::decimal256(10, 9),

python\_decimal, -1000000000));

return Status::OK();

}

Status TestDecimal128FromPythonInteger() {

Decimal128 value;

OwnedRef python\_long(PyLong\_FromLong(42));

auto type = ::arrow::decimal128(10, 2);

const auto& decimal\_type = checked\_cast<const DecimalType&>(\*type);

ASSERT\_OK(internal::DecimalFromPyObject(python\_long.obj(), decimal\_type, &value));

ASSERT\_EQ(4200, value);

return Status::OK();

}

Status TestDecimal256FromPythonInteger() {

Decimal256 value;

OwnedRef python\_long(PyLong\_FromLong(42));

auto type = ::arrow::decimal256(10, 2);

const auto& decimal\_type = checked\_cast<const DecimalType&>(\*type);

ASSERT\_OK(internal::DecimalFromPyObject(python\_long.obj(), decimal\_type, &value));

ASSERT\_EQ(4200, value);

return Status::OK();

}

Status TestDecimal128OverflowFails() {

Decimal128 value;

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("9999999999999999999999999999999999999.9");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

internal::DecimalMetadata metadata;

ASSERT\_OK(metadata.Update(python\_decimal));

ASSERT\_EQ(38, metadata.precision());

ASSERT\_EQ(1, metadata.scale());

auto type = ::arrow::decimal(38, 38);

const auto& decimal\_type = checked\_cast<const DecimalType&>(\*type);

ASSERT\_RAISES(Invalid,

internal::DecimalFromPythonDecimal(python\_decimal, decimal\_type, &value));

return Status::OK();

}

Status TestDecimal256OverflowFails() {

Decimal256 value;

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string(

"999999999999999999999999999999999999999999999999999999999999999999999999999.9");

PyObject\* python\_decimal =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

internal::DecimalMetadata metadata;

ASSERT\_OK(metadata.Update(python\_decimal));

ASSERT\_EQ(76, metadata.precision());

ASSERT\_EQ(1, metadata.scale());

auto type = ::arrow::decimal(76, 76);

const auto& decimal\_type = checked\_cast<const DecimalType&>(\*type);

ASSERT\_RAISES(Invalid,

internal::DecimalFromPythonDecimal(python\_decimal, decimal\_type, &value));

return Status::OK();

}

Status TestNoneAndNaN() {

OwnedRef list\_ref(PyList\_New(4));

PyObject\* list = list\_ref.obj();

ASSERT\_NE(list, nullptr);

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

PyObject\* constructor = decimal\_constructor\_.obj();

PyObject\* decimal\_value = internal::DecimalFromString(constructor, "1.234");

ASSERT\_NE(decimal\_value, nullptr);

Py\_INCREF(Py\_None);

PyObject\* missing\_value1 = Py\_None;

ASSERT\_NE(missing\_value1, nullptr);

PyObject\* missing\_value2 = PyFloat\_FromDouble(NPY\_NAN);

ASSERT\_NE(missing\_value2, nullptr);

PyObject\* missing\_value3 = internal::DecimalFromString(constructor, "nan");

ASSERT\_NE(missing\_value3, nullptr);

// This steals a reference to each object, so we don't need to decref them later,

// just the list

ASSERT\_EQ(0, PyList\_SetItem(list, 0, decimal\_value));

ASSERT\_EQ(0, PyList\_SetItem(list, 1, missing\_value1));

ASSERT\_EQ(0, PyList\_SetItem(list, 2, missing\_value2));

ASSERT\_EQ(0, PyList\_SetItem(list, 3, missing\_value3));

PyConversionOptions options;

ASSERT\_RAISES(TypeError, ConvertPySequence(list, nullptr, options));

options.from\_pandas = true;

auto chunked = std::move(ConvertPySequence(list, nullptr, options)).ValueOrDie();

ASSERT\_EQ(chunked->num\_chunks(), 1);

auto arr = chunked->chunk(0);

ASSERT\_TRUE(arr->IsValid(0));

ASSERT\_TRUE(arr->IsNull(1));

ASSERT\_TRUE(arr->IsNull(2));

ASSERT\_TRUE(arr->IsNull(3));

return Status::OK();

}

Status TestMixedPrecisionAndScale() {

std::vector<std::string> strings{{"0.001", "1.01E5", "1.01E5"}};

OwnedRef list\_ref(PyList\_New(static\_cast<Py\_ssize\_t>(strings.size())));

PyObject\* list = list\_ref.obj();

ASSERT\_NE(list, nullptr);

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

// PyList\_SetItem steals a reference to the item so we don't decref it later

PyObject\* decimal\_constructor = decimal\_constructor\_.obj();

for (Py\_ssize\_t i = 0; i < static\_cast<Py\_ssize\_t>(strings.size()); ++i) {

const int result = PyList\_SetItem(

list, i, internal::DecimalFromString(decimal\_constructor, strings.at(i)));

ASSERT\_EQ(0, result);

}

auto arr = std::move(ConvertPySequence(list, nullptr, {})).ValueOrDie();

const auto& type = checked\_cast<const DecimalType&>(\*arr->type());

int32\_t expected\_precision = 9;

int32\_t expected\_scale = 3;

ASSERT\_EQ(expected\_precision, type.precision());

ASSERT\_EQ(expected\_scale, type.scale());

return Status::OK();

}

Status TestMixedPrecisionAndScaleSequenceConvert() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string\_1("0.01");

PyObject\* value1 =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string\_1);

ASSERT\_NE(value1, nullptr);

std::string decimal\_string\_2("0.001");

PyObject\* value2 =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string\_2);

ASSERT\_NE(value2, nullptr);

OwnedRef list\_ref(PyList\_New(2));

PyObject\* list = list\_ref.obj();

// This steals a reference to each object, so we don't need to decref them later

// just the list

ASSERT\_EQ(PyList\_SetItem(list, 0, value1), 0);

ASSERT\_EQ(PyList\_SetItem(list, 1, value2), 0);

auto arr = std::move(ConvertPySequence(list, nullptr, {})).ValueOrDie();

const auto& type = checked\_cast<const Decimal128Type&>(\*arr->type());

ASSERT\_EQ(3, type.precision());

ASSERT\_EQ(3, type.scale());

return Status::OK();

}

Status TestSimpleInference() {

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("0.01");

PyObject\* value =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

ASSERT\_NE(value, nullptr);

internal::DecimalMetadata metadata;

ASSERT\_OK(metadata.Update(value));

ASSERT\_EQ(2, metadata.precision());

ASSERT\_EQ(2, metadata.scale());

return Status::OK();

}

Status TestUpdateWithNaN() {

internal::DecimalMetadata metadata;

OwnedRef decimal\_constructor\_;

OwnedRef decimal\_module;

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

RETURN\_NOT\_OK(

internal::ImportFromModule(decimal\_module.obj(), "Decimal", &decimal\_constructor\_));

std::string decimal\_string("nan");

PyObject\* nan\_value =

internal::DecimalFromString(decimal\_constructor\_.obj(), decimal\_string);

ASSERT\_OK(metadata.Update(nan\_value));

ASSERT\_EQ(std::numeric\_limits<int32\_t>::min(), metadata.precision());

ASSERT\_EQ(std::numeric\_limits<int32\_t>::min(), metadata.scale());

return Status::OK();

}

} // namespace

std::vector<TestCase> GetCppTestCases() {

return {

{"test\_owned\_ref\_moves", TestOwnedRefMoves},

{"test\_owned\_ref\_nogil\_moves", TestOwnedRefNoGILMoves},

{"test\_check\_pyerror\_status", TestCheckPyErrorStatus},

{"test\_check\_pyerror\_status\_nogil", TestCheckPyErrorStatusNoGIL},

{"test\_restore\_pyerror\_basics", TestRestorePyErrorBasics},

{"test\_pybuffer\_invalid\_input\_object", TestPyBufferInvalidInputObject},

#ifndef \_WIN32

{"test\_pybuffer\_numpy\_array", TestPyBufferNumpyArray},

{"test\_numpybuffer\_numpy\_array", TestNumPyBufferNumpyArray},

#endif

{"test\_python\_decimal\_to\_string", TestPythonDecimalToString},

{"test\_infer\_precision\_and\_scale", TestInferPrecisionAndScale},

{"test\_infer\_precision\_and\_negative\_scale", TestInferPrecisionAndNegativeScale},

{"test\_infer\_all\_leading\_zeros", TestInferAllLeadingZeros},

{"test\_infer\_all\_leading\_zeros\_exponential\_notation\_positive",

TestInferAllLeadingZerosExponentialNotationPositive},

{"test\_infer\_all\_leading\_zeros\_exponential\_notation\_negative",

TestInferAllLeadingZerosExponentialNotationNegative},

{"test\_object\_block\_write\_fails\_pandas\_convert", TestObjectBlockWriteFails},

{"test\_mixed\_type\_fails", TestMixedTypeFails},

{"test\_from\_python\_decimal\_rescale\_not\_truncateable",

TestFromPythonDecimalRescaleNotTruncateable},

{"test\_from\_python\_decimal\_rescale\_truncateable",

TestFromPythonDecimalRescaleTruncateable},

{"test\_from\_python\_negative\_decimal\_rescale", TestFromPythonNegativeDecimalRescale},

{"test\_decimal128\_from\_python\_integer", TestDecimal128FromPythonInteger},

{"test\_decimal256\_from\_python\_integer", TestDecimal256FromPythonInteger},

{"test\_decimal128\_overflow\_fails", TestDecimal128OverflowFails},

{"test\_decimal256\_overflow\_fails", TestDecimal256OverflowFails},

{"test\_none\_and\_nan", TestNoneAndNaN},

{"test\_mixed\_precision\_and\_scale", TestMixedPrecisionAndScale},

{"test\_mixed\_precision\_and\_scale\_sequence\_convert",

TestMixedPrecisionAndScaleSequenceConvert},

{"test\_simple\_inference", TestSimpleInference},

{"test\_update\_with\_nan", TestUpdateWithNaN},

};

}

} // namespace testing

} // namespace py

} // namespace arrow