// 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.

// helpers.h includes a NumPy header, so we include this first

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

#include "arrow/python/helpers.h"

#include <cmath>

#include <limits>

#include <mutex>

#include <sstream>

#include <type\_traits>

#include "arrow/python/common.h"

#include "arrow/python/decimal.h"

#include "arrow/type\_fwd.h"

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

#include "arrow/util/config.h"

#include "arrow/util/logging.h"

namespace arrow {

using internal::checked\_cast;

namespace py {

#define GET\_PRIMITIVE\_TYPE(NAME, FACTORY) \

case Type::NAME: \

return FACTORY()

std::shared\_ptr<DataType> GetPrimitiveType(Type::type type) {

switch (type) {

case Type::NA:

return null();

GET\_PRIMITIVE\_TYPE(UINT8, uint8);

GET\_PRIMITIVE\_TYPE(INT8, int8);

GET\_PRIMITIVE\_TYPE(UINT16, uint16);

GET\_PRIMITIVE\_TYPE(INT16, int16);

GET\_PRIMITIVE\_TYPE(UINT32, uint32);

GET\_PRIMITIVE\_TYPE(INT32, int32);

GET\_PRIMITIVE\_TYPE(UINT64, uint64);

GET\_PRIMITIVE\_TYPE(INT64, int64);

GET\_PRIMITIVE\_TYPE(DATE32, date32);

GET\_PRIMITIVE\_TYPE(DATE64, date64);

GET\_PRIMITIVE\_TYPE(BOOL, boolean);

GET\_PRIMITIVE\_TYPE(HALF\_FLOAT, float16);

GET\_PRIMITIVE\_TYPE(FLOAT, float32);

GET\_PRIMITIVE\_TYPE(DOUBLE, float64);

GET\_PRIMITIVE\_TYPE(BINARY, binary);

GET\_PRIMITIVE\_TYPE(STRING, utf8);

GET\_PRIMITIVE\_TYPE(LARGE\_BINARY, large\_binary);

GET\_PRIMITIVE\_TYPE(LARGE\_STRING, large\_utf8);

GET\_PRIMITIVE\_TYPE(BINARY\_VIEW, binary\_view);

GET\_PRIMITIVE\_TYPE(STRING\_VIEW, utf8\_view);

GET\_PRIMITIVE\_TYPE(INTERVAL\_MONTH\_DAY\_NANO, month\_day\_nano\_interval);

default:

return nullptr;

}

}

PyObject\* PyHalf\_FromHalf(npy\_half value) {

PyObject\* result = PyArrayScalar\_New(Half);

if (result != NULL) {

PyArrayScalar\_ASSIGN(result, Half, value);

}

return result;

}

Status PyFloat\_AsHalf(PyObject\* obj, npy\_half\* out) {

if (PyArray\_IsScalar(obj, Half)) {

\*out = PyArrayScalar\_VAL(obj, Half);

return Status::OK();

} else {

// XXX: cannot use npy\_double\_to\_half() without linking with Numpy

return Status::TypeError("Expected np.float16 instance");

}

}

namespace internal {

std::string PyBytes\_AsStdString(PyObject\* obj) {

DCHECK(PyBytes\_Check(obj));

return std::string(PyBytes\_AS\_STRING(obj), PyBytes\_GET\_SIZE(obj));

}

Status PyUnicode\_AsStdString(PyObject\* obj, std::string\* out) {

DCHECK(PyUnicode\_Check(obj));

Py\_ssize\_t size;

// The utf-8 representation is cached on the unicode object

const char\* data = PyUnicode\_AsUTF8AndSize(obj, &size);

RETURN\_IF\_PYERROR();

\*out = std::string(data, size);

return Status::OK();

}

std::string PyObject\_StdStringRepr(PyObject\* obj) {

OwnedRef unicode\_ref(PyObject\_Repr(obj));

OwnedRef bytes\_ref;

if (unicode\_ref) {

bytes\_ref.reset(

PyUnicode\_AsEncodedString(unicode\_ref.obj(), "utf8", "backslashreplace"));

}

if (!bytes\_ref) {

PyErr\_Clear();

std::stringstream ss;

ss << "<object of type '" << Py\_TYPE(obj)->tp\_name << "' repr() failed>";

return ss.str();

}

return PyBytes\_AsStdString(bytes\_ref.obj());

}

Status PyObject\_StdStringStr(PyObject\* obj, std::string\* out) {

OwnedRef string\_ref(PyObject\_Str(obj));

RETURN\_IF\_PYERROR();

return PyUnicode\_AsStdString(string\_ref.obj(), out);

}

Result<bool> IsModuleImported(const std::string& module\_name) {

// PyImport\_GetModuleDict returns with a borrowed reference

OwnedRef key(PyUnicode\_FromString(module\_name.c\_str()));

auto is\_imported = PyDict\_Contains(PyImport\_GetModuleDict(), key.obj());

RETURN\_IF\_PYERROR();

return is\_imported;

}

Status ImportModule(const std::string& module\_name, OwnedRef\* ref) {

PyObject\* module = PyImport\_ImportModule(module\_name.c\_str());

RETURN\_IF\_PYERROR();

ref->reset(module);

return Status::OK();

}

Status ImportFromModule(PyObject\* module, const std::string& name, OwnedRef\* ref) {

PyObject\* attr = PyObject\_GetAttrString(module, name.c\_str());

RETURN\_IF\_PYERROR();

ref->reset(attr);

return Status::OK();

}

namespace {

Status IntegerOverflowStatus(PyObject\* obj, const std::string& overflow\_message) {

if (overflow\_message.empty()) {

std::string obj\_as\_stdstring;

RETURN\_NOT\_OK(PyObject\_StdStringStr(obj, &obj\_as\_stdstring));

return Status::Invalid("Value ", obj\_as\_stdstring,

" too large to fit in C integer type");

} else {

return Status::Invalid(overflow\_message);

}

}

Result<OwnedRef> PyObjectToPyInt(PyObject\* obj) {

// Try to call \_\_index\_\_ or \_\_int\_\_ on `obj`

// (starting from Python 3.10, the latter isn't done anymore by PyLong\_AsLong\*).

OwnedRef ref(PyNumber\_Index(obj));

if (ref) {

return std::move(ref);

}

PyErr\_Clear();

const auto nb = Py\_TYPE(obj)->tp\_as\_number;

if (nb && nb->nb\_int) {

ref.reset(nb->nb\_int(obj));

if (!ref) {

RETURN\_IF\_PYERROR();

}

DCHECK(ref);

return std::move(ref);

}

return Status::TypeError(

"object of type ",

PyObject\_StdStringRepr(reinterpret\_cast<PyObject\*>(Py\_TYPE(obj))),

" cannot be converted to int");

}

// Extract C signed int from Python object

template <typename Int, enable\_if\_t<std::is\_signed<Int>::value, Int> = 0>

Status CIntFromPythonImpl(PyObject\* obj, Int\* out, const std::string& overflow\_message) {

static\_assert(sizeof(Int) <= sizeof(long long), // NOLINT

"integer type larger than long long");

OwnedRef ref;

if (!PyLong\_Check(obj)) {

ARROW\_ASSIGN\_OR\_RAISE(ref, PyObjectToPyInt(obj));

obj = ref.obj();

}

if (sizeof(Int) > sizeof(long)) { // NOLINT

const auto value = PyLong\_AsLongLong(obj);

if (ARROW\_PREDICT\_FALSE(value == -1)) {

RETURN\_IF\_PYERROR();

}

if (ARROW\_PREDICT\_FALSE(value < std::numeric\_limits<Int>::min() ||

value > std::numeric\_limits<Int>::max())) {

return IntegerOverflowStatus(obj, overflow\_message);

}

\*out = static\_cast<Int>(value);

} else {

const auto value = PyLong\_AsLong(obj);

if (ARROW\_PREDICT\_FALSE(value == -1)) {

RETURN\_IF\_PYERROR();

}

if (ARROW\_PREDICT\_FALSE(value < std::numeric\_limits<Int>::min() ||

value > std::numeric\_limits<Int>::max())) {

return IntegerOverflowStatus(obj, overflow\_message);

}

\*out = static\_cast<Int>(value);

}

return Status::OK();

}

// Extract C unsigned int from Python object

template <typename Int, enable\_if\_t<std::is\_unsigned<Int>::value, Int> = 0>

Status CIntFromPythonImpl(PyObject\* obj, Int\* out, const std::string& overflow\_message) {

static\_assert(sizeof(Int) <= sizeof(unsigned long long), // NOLINT

"integer type larger than unsigned long long");

OwnedRef ref;

if (!PyLong\_Check(obj)) {

ARROW\_ASSIGN\_OR\_RAISE(ref, PyObjectToPyInt(obj));

obj = ref.obj();

}

if (sizeof(Int) > sizeof(unsigned long)) { // NOLINT

const auto value = PyLong\_AsUnsignedLongLong(obj);

if (ARROW\_PREDICT\_FALSE(value == static\_cast<decltype(value)>(-1))) {

RETURN\_IF\_PYERROR();

}

if (ARROW\_PREDICT\_FALSE(value > std::numeric\_limits<Int>::max())) {

return IntegerOverflowStatus(obj, overflow\_message);

}

\*out = static\_cast<Int>(value);

} else {

const auto value = PyLong\_AsUnsignedLong(obj);

if (ARROW\_PREDICT\_FALSE(value == static\_cast<decltype(value)>(-1))) {

RETURN\_IF\_PYERROR();

}

if (ARROW\_PREDICT\_FALSE(value > std::numeric\_limits<Int>::max())) {

return IntegerOverflowStatus(obj, overflow\_message);

}

\*out = static\_cast<Int>(value);

}

return Status::OK();

}

} // namespace

template <typename Int>

Status CIntFromPython(PyObject\* obj, Int\* out, const std::string& overflow\_message) {

if (PyBool\_Check(obj)) {

return Status::TypeError("Expected integer, got bool");

}

return CIntFromPythonImpl(obj, out, overflow\_message);

}

template Status CIntFromPython(PyObject\*, int8\_t\*, const std::string&);

template Status CIntFromPython(PyObject\*, int16\_t\*, const std::string&);

template Status CIntFromPython(PyObject\*, int32\_t\*, const std::string&);

template Status CIntFromPython(PyObject\*, int64\_t\*, const std::string&);

template Status CIntFromPython(PyObject\*, uint8\_t\*, const std::string&);

template Status CIntFromPython(PyObject\*, uint16\_t\*, const std::string&);

template Status CIntFromPython(PyObject\*, uint32\_t\*, const std::string&);

template Status CIntFromPython(PyObject\*, uint64\_t\*, const std::string&);

inline bool MayHaveNaN(PyObject\* obj) {

// Some core types can be very quickly type-checked and do not allow NaN values

const int64\_t non\_nan\_tpflags = Py\_TPFLAGS\_LONG\_SUBCLASS | Py\_TPFLAGS\_LIST\_SUBCLASS |

Py\_TPFLAGS\_TUPLE\_SUBCLASS | Py\_TPFLAGS\_BYTES\_SUBCLASS |

Py\_TPFLAGS\_UNICODE\_SUBCLASS | Py\_TPFLAGS\_DICT\_SUBCLASS |

Py\_TPFLAGS\_BASE\_EXC\_SUBCLASS | Py\_TPFLAGS\_TYPE\_SUBCLASS;

return !PyType\_HasFeature(Py\_TYPE(obj), non\_nan\_tpflags);

}

bool PyFloat\_IsNaN(PyObject\* obj) {

return PyFloat\_Check(obj) && std::isnan(PyFloat\_AsDouble(obj));

}

namespace {

// This needs a conditional, because using std::once\_flag could introduce

// a deadlock when the GIL is enabled. See

// https://github.com/apache/arrow/commit/f69061935e92e36e25bb891177ca8bc4f463b272 for

// more info.

#ifdef Py\_GIL\_DISABLED

static std::once\_flag pandas\_static\_initialized;

#else

static bool pandas\_static\_initialized = false;

#endif

// Once initialized, these variables hold borrowed references to Pandas static data.

// We should not use OwnedRef here because Python destructors would be

// called on a finalized interpreter.

static PyObject\* pandas\_NA = nullptr;

static PyObject\* pandas\_NaT = nullptr;

static PyObject\* pandas\_Timedelta = nullptr;

static PyObject\* pandas\_Timestamp = nullptr;

static PyTypeObject\* pandas\_NaTType = nullptr;

static PyObject\* pandas\_DateOffset = nullptr;

void GetPandasStaticSymbols() {

OwnedRef pandas;

// Import pandas

Status s = ImportModule("pandas", &pandas);

if (!s.ok()) {

return;

}

#ifndef Py\_GIL\_DISABLED

// Since ImportModule can release the GIL, another thread could have

// already initialized the static data.

if (pandas\_static\_initialized) {

return;

}

#endif

OwnedRef ref;

// set NaT sentinel and its type

if (ImportFromModule(pandas.obj(), "NaT", &ref).ok()) {

pandas\_NaT = ref.obj();

// PyObject\_Type returns a new reference but we trust that pandas.NaT will

// outlive our use of this PyObject\*

pandas\_NaTType = Py\_TYPE(ref.obj());

}

// retain a reference to Timedelta

if (ImportFromModule(pandas.obj(), "Timedelta", &ref).ok()) {

pandas\_Timedelta = ref.obj();

}

// retain a reference to Timestamp

if (ImportFromModule(pandas.obj(), "Timestamp", &ref).ok()) {

pandas\_Timestamp = ref.obj();

}

// if pandas.NA exists, retain a reference to it

if (ImportFromModule(pandas.obj(), "NA", &ref).ok()) {

pandas\_NA = ref.obj();

}

// Import DateOffset type

if (ImportFromModule(pandas.obj(), "DateOffset", &ref).ok()) {

pandas\_DateOffset = ref.obj();

}

}

} // namespace

#ifdef Py\_GIL\_DISABLED

void InitPandasStaticData() {

std::call\_once(pandas\_static\_initialized, GetPandasStaticSymbols);

}

#else

void InitPandasStaticData() {

// NOTE: This is called with the GIL held. We needn't (and shouldn't,

// to avoid deadlocks) use an additional C++ lock (ARROW-10519).

if (pandas\_static\_initialized) {

return;

}

GetPandasStaticSymbols();

pandas\_static\_initialized = true;

}

#endif

bool PandasObjectIsNull(PyObject\* obj) {

if (!MayHaveNaN(obj)) {

return false;

}

if (obj == Py\_None) {

return true;

}

if (PyFloat\_IsNaN(obj) || (pandas\_NA && obj == pandas\_NA) ||

(pandas\_NaTType && PyObject\_TypeCheck(obj, pandas\_NaTType)) ||

(internal::PyDecimal\_Check(obj) && internal::PyDecimal\_ISNAN(obj))) {

return true;

}

return false;

}

bool IsPandasTimedelta(PyObject\* obj) {

return pandas\_Timedelta && PyObject\_IsInstance(obj, pandas\_Timedelta);

}

bool IsPandasTimestamp(PyObject\* obj) {

return pandas\_Timestamp && PyObject\_IsInstance(obj, pandas\_Timestamp);

}

PyObject\* BorrowPandasDataOffsetType() { return pandas\_DateOffset; }

Status InvalidValue(PyObject\* obj, const std::string& why) {

auto obj\_as\_str = PyObject\_StdStringRepr(obj);

return Status::Invalid("Could not convert ", std::move(obj\_as\_str), " with type ",

Py\_TYPE(obj)->tp\_name, ": ", why);

}

Status InvalidType(PyObject\* obj, const std::string& why) {

auto obj\_as\_str = PyObject\_StdStringRepr(obj);

return Status::TypeError("Could not convert ", std::move(obj\_as\_str), " with type ",

Py\_TYPE(obj)->tp\_name, ": ", why);

}

Status UnboxIntegerAsInt64(PyObject\* obj, int64\_t\* out) {

if (PyLong\_Check(obj)) {

int overflow = 0;

\*out = PyLong\_AsLongLongAndOverflow(obj, &overflow);

if (overflow) {

return Status::Invalid("PyLong is too large to fit int64");

}

} else if (PyArray\_IsScalar(obj, Byte)) {

\*out = reinterpret\_cast<PyByteScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, UByte)) {

\*out = reinterpret\_cast<PyUByteScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, Short)) {

\*out = reinterpret\_cast<PyShortScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, UShort)) {

\*out = reinterpret\_cast<PyUShortScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, Int)) {

\*out = reinterpret\_cast<PyIntScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, UInt)) {

\*out = reinterpret\_cast<PyUIntScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, Long)) {

\*out = reinterpret\_cast<PyLongScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, ULong)) {

\*out = reinterpret\_cast<PyULongScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, LongLong)) {

\*out = reinterpret\_cast<PyLongLongScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, Int64)) {

\*out = reinterpret\_cast<PyInt64ScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, ULongLong)) {

\*out = reinterpret\_cast<PyULongLongScalarObject\*>(obj)->obval;

} else if (PyArray\_IsScalar(obj, UInt64)) {

\*out = reinterpret\_cast<PyUInt64ScalarObject\*>(obj)->obval;

} else {

return Status::Invalid("Integer scalar type not recognized");

}

return Status::OK();

}

Status IntegerScalarToDoubleSafe(PyObject\* obj, double\* out) {

int64\_t value = 0;

RETURN\_NOT\_OK(UnboxIntegerAsInt64(obj, &value));

constexpr int64\_t kDoubleMax = 1LL << 53;

constexpr int64\_t kDoubleMin = -(1LL << 53);

if (value < kDoubleMin || value > kDoubleMax) {

return Status::Invalid("Integer value ", value, " is outside of the range exactly",

" representable by a IEEE 754 double precision value");

}

\*out = static\_cast<double>(value);

return Status::OK();

}

Status IntegerScalarToFloat32Safe(PyObject\* obj, float\* out) {

int64\_t value = 0;

RETURN\_NOT\_OK(UnboxIntegerAsInt64(obj, &value));

constexpr int64\_t kFloatMax = 1LL << 24;

constexpr int64\_t kFloatMin = -(1LL << 24);

if (value < kFloatMin || value > kFloatMax) {

return Status::Invalid("Integer value ", value, " is outside of the range exactly",

" representable by a IEEE 754 single precision value");

}

\*out = static\_cast<float>(value);

return Status::OK();

}

void DebugPrint(PyObject\* obj) {

std::string repr = PyObject\_StdStringRepr(obj);

PySys\_WriteStderr("%s\n", repr.c\_str());

}

bool IsThreadingEnabled() {

#ifdef ARROW\_ENABLE\_THREADING

return true;

#else

return false;

#endif

}

} // namespace internal

} // namespace py

} // namespace arrow