

---

# **xtensor-python**

**Dec 05, 2018**



---

# INSTALLATION

---

<b>1</b>	<b>Enabling numpy arrays in your C++ libraries</b>	<b>3</b>
<b>2</b>	<b>Licensing</b>	<b>5</b>
2.1	Installation . . . . .	5
2.2	Basic Usage . . . . .	6
2.3	Arrays and tensors . . . . .	7
2.4	Importing numpy C API . . . . .	8
2.5	Getting started with xtensor-python-cookiecutter . . . . .	9
2.6	API reference . . . . .	10
2.7	Releasing xtensor-python . . . . .	14



Python bindings for the `xtensor` C++ multi-dimensional array library.

What are `xtensor` and `xtensor-python`?

- `xtensor` is a C++ library for multi-dimensional arrays enabling numpy-style broadcasting and lazy computing.
- `xtensor-python` enables inplace use of numpy arrays with all the benefits from `xtensor`
  - C++ universal functions and broadcasting
  - STL - compliant APIs.

The [numpy to xtensor cheat sheet](#) from the `xtensor` documentation shows how numpy APIs translate to C++ with `xtensor`.

The Python bindings for `xtensor` are based on the `pybind11` C++ library, which enables seamless interoperability between C++ and Python.



---

## Enabling numpy arrays in your C++ libraries

---

Instead of exposing new types to python, `xtensor-python` enables the use of [NumPy](#) data structures from C++ using Python's [Buffer Protocol](#).

In addition to the basic accessors and iterators of `xtensor` containers, it also enables using numpy arrays with `xtensor`'s expression system.

Besides `xtensor-python` provides an API to create *Universal functions* from simple scalar functions from your C++ code.

Finally, a cookiecutter template project is provided. It takes care of the initial work of generating a project skeleton for a C++ extension based on `xtensor-python` containing a few examples, unit tests and HTML documentation. Find out more about the [xtensor-python-cookiecutter](#).

`xtensor` and `xtensor-python` require a modern C++ compiler supporting C++14. The following C++ compilers are supported:

- On Windows platforms, Visual C++ 2015 Update 2, or more recent
- On Unix platforms, gcc 4.9 or a recent version of Clang





We use a shared copyright model that enables all contributors to maintain the copyright on their contributions. This software is licensed under the BSD-3-Clause license. See the LICENSE file for details.

## 2.1 Installation

Although `xtensor-python` is a header-only library, we provide standardized means to install it, with package managers or with `cmake`.

Besides the `xtensor-python` headers, all these methods place the `cmake` project configuration file in the right location so that third-party projects can use `cmake`'s `find_package` to locate `xtensor-python` headers.

### 2.1.1 Using the conda package

A package for `xtensor-python` is available on the conda package manager.

```
conda install -c conda-forge xtensor-python
```

### 2.1.2 Using the Debian package

A package for `xtensor-python` is available on Debian.

```
sudo apt-get install xtensor-python-dev
```

### 2.1.3 From source with cmake

You can also install `xtensor-python` from source with `cmake`. On Unix platforms, from the source directory:

```
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/path/to/prefix ..
make install
```

On Windows platforms, from the source directory:

```
mkdir build
cd build
cmake -G "NMake Makefiles" -DCMAKE_INSTALL_PREFIX=/path/to/prefix ..
nmake
nmake install
```

See the section of the documentation on build-options, for more details on how to `cmake` options.

## 2.2 Basic Usage

### 2.2.1 Example 1: Use an algorithm of the C++ library on a numpy array inplace

**C++ code**

```
#include <numeric> // Standard library import for_
↳std::accumulate
#include "pybind11/pybind11.h" // Pybind11 import to define Python bindings
#include "xtensor/xmath.hpp" // xtensor import for the C++ universal_
↳functions
#define FORCE_IMPORT_ARRAY // numpy C api loading
#include "xtensor-python/pyarray.hpp" // Numpy bindings

double sum_of_sines(xt::pyarray<double>& m)
{
    auto sines = xt::sin(m); // sines does not actually hold values.
    return std::accumulate(sines.cbegin(), sines.cend(), 0.0);
}

PYBIND11_MODULE(xtensor_python_test, m)
{
    xt::import_numpy();
    m.doc() = "Test module for xtensor python bindings";

    m.def("sum_of_sines", sum_of_sines, "Sum the sines of the input values");
}
```

**Python code:**

```
import numpy as np
import xtensor_python_test as xt

a = np.arange(15).reshape(3, 5)
s = xt.sum_of_sines(v)
s
```

**Outputs**

```
1.2853996391883833
```

**2.2.2 Example 2: Create a numpy-style universal function from a C++ scalar function****C++ code**

```
#include "pybind11/pybind11.h"
#define FORCE_IMPORT_ARRAY
#include "xtensor-python/pyvectorize.hpp"
#include <numeric>
#include <cmath>

namespace py = pybind11;

double scalar_func(double i, double j)
{
    return std::sin(i) - std::cos(j);
}

PYBIND11_MODULE(xtensor_python_test, m)
{
    xt::import_numpy();
    m.doc() = "Test module for xtensor python bindings";

    m.def("vectorized_func", xt::pyvectorize(scalar_func), "");
}
```

**Python code:**

```
import numpy as np
import xtensor_python_test as xt

x = np.arange(15).reshape(3, 5)
y = [1, 2, 3, 4, 5]
z = xt.vectorized_func(x, y)
z
```

**Outputs**

```
[[-0.540302,  1.257618,  1.89929 ,  0.794764, -1.040465],
 [-1.499227,  0.136731,  1.646979,  1.643002,  0.128456],
 [-1.084323, -0.583843,  0.45342 ,  1.073811,  0.706945]]
```

**2.3 Arrays and tensors**

xtensor-python provides two container types wrapping numpy arrays: pyarray and pytensor. They are the counterparts to xarray and xtensor containers.

### 2.3.1 pyarray

Like `xarray`, `pyarray` has a dynamic shape. This means that you can reshape the numpy array on the C++ side and see this change reflected on the python side. `pyarray` doesn't make a copy of the shape or the strides, but reads them each time it is needed. Therefore, if a reference on a `pyarray` is kept in the C++ code and the corresponding numpy array is then reshaped in the python code, this modification will reflect in the `pyarray`.

### 2.3.2 pytensor

Like `xtensor`, `pytensor` has a static stack-allocated shape. This means that the shape of the numpy array is copied into the shape of the `pytensor` upon creation. As a consequence, reshapes are not reflected across languages. However, this drawback is offset by a more effective computation of shape and broadcast.

## 2.4 Importing numpy C API

Importing the C API module of numpy requires more code than just including a header. `xtensor-python` simplifies a lot this import, however some actions are still required in the user code.

### 2.4.1 Extension module with a single file

When writing an extension module that is self-contained in a single file, its author should pay attention to the following points:

- `FORCE_IMPORT_ARRAY` must be defined before including any header of `xtensor-python`.
- `xt::import_numpy()` must be called in the function initializing the module.

Thus the basic skeleton of the module looks like:

```
#define FORCE_IMPORT_ARRAY
#include "xtensor-python/pyarray.hpp"

PYBIND11_MODULE(plugin_name, m)
{
    xt::import_numpy();
    //...
}
```

### 2.4.2 Extension module with multiple files

If the extension module contains many source files that include `xtensor-python` header files, the previous points are still required. However, the symbol `FORCE_IMPORT_ARRAY` must be defined only once. The simplest is to define it in the file that contains the initializing code of the module, you can then directly include `xtensor-python` headers in other files. Let's illustrate this with an extension modules containing the following files:

- `main.cpp`: initializing code of the module
- `image.hpp`: declaration of the `image` class embedding an `xt::pyarray` object
- `image.cpp`: implementation of the `image` class

The basic skeleton of the module looks like:

```

// image.hpp
// Do NOT define FORCE_IMPORT_ARRAY here
#include "xtensor-python/pyarray.hpp"

class image
{
// ....
private:
    xt::pyarray<double> m_data;
};

// image.cpp
// Do NOT define FORCE_IMPORT_ARRAY here
#include "image.hpp"
// definition of the image class

// main.cpp
// FORCE_IMPORT_ARRAY must be define ONCE, BEFORE including
// any header from xtensor-python (even indirectly)
#define FORCE_IMPORT_ARRAY
#include "image.hpp"
PYBIND11_MODULE(plugin_name, m)
{
    xt::import_numpy();
    //...
}

```

### 2.4.3 Using other extension modules

Including an header of `xtensor-python` actually defines `PY_ARRAY_UNIQUE_SYMBOL` to `xtensor_python_ARRAY_API`. This might be problematic if you import another library that defines its own `PY_ARRAY_UNIQUE_SYMBOL`, or if you define yours. If so, you can override the behavior of `xtensor-python` by explicitly defining `PY_ARRAY_UNIQUE_SYMBOL` prior to including any `xtensor-python` header:

```

// in every source file
#define PY_ARRAY_UNIQUE_SYMBOL my_uniqe_array_api
#include "xtensor-python/pyarray.hpp"

```

## 2.5 Getting started with `xtensor-python-cookiecutter`

`xtensor-python-cookiecutter` helps extension authors create Python extension modules making use of `xtensor`.

It takes care of the initial work of generating a project skeleton with

- A complete `setup.py` compiling the extension module
- A few examples included in the resulting project including
  - A universal function defined from C++
  - A function making use of an algorithm from the STL on a numpy array
  - Unit tests
  - The generation of the HTML documentation with sphinx

## 2.5.1 Usage

Install `cookiecutter`

```
pip install cookiecutter
```

After installing `cookiecutter`, use the `xtensor-python-cookiecutter`:

```
cookiecutter https://github.com/QuantStack/xtensor-python-cookiecutter.git
```

As `xtensor-python-cookiecutter` runs, you will be asked for basic information about your custom extension project. You will be prompted for the following information:

- `author_name`: your name or the name of your organization,
- `author_email`: your project's contact email,
- `github_project_name`: name of the GitHub repository for your project,
- `github_organization_name`: name of the GitHub organization for your project,
- `python_package_name`: name of the Python package created by your extension,
- `cpp_namespace`: name for the C++ namespace holding the implementation of your extension,
- `project_short_description`: a short description for your project.

This will produce a directory containing all the required content for a minimal extension project making use of `xtensor` with all the required boilerplate for package management, together with a few basic examples.

## 2.6 API reference

### 2.6.1 Containers

#### `pyarray`

**template** <class *T*, layout\_type *L*>

**class** `pyarray`: **public** `xt::pycontainer<pyarray<T, L>>`, **public** `xcontainer_semantic<pyarray<T, L>>`

Multidimensional container providing the `xtensor` container semantics to a numpy array.

`pyarray` is similar to the `xarray` container in that it has a dynamic dimensionality. Reshapes of a `pyarray` container are reflected in the underlying numpy array.

See *pytensor*

#### Template Parameters

- `T`: The type of the element stored in the `pyarray`.

#### Constructors

`pyarray` (**const** value\_type &*t*)

Allocates a `pyarray` with nested initializer lists.

`pyarray` (**const** shape\_type &*shape*, layout\_type *l* = layout\_type::row\_major)

Allocates an uninitialized `pyarray` with the specified shape and layout.

**Parameters**

- `shape`: the shape of the pyarray
- `l`: the layout of the pyarray

**pyarray** (`const` `shape_type` *&shape*, `const_reference` *value*, `layout_type` *l* = `layout_type::row_major`)

Allocates a pyarray with the specified shape and layout.

Elements are initialized to the specified value.

**Parameters**

- `shape`: the shape of the pyarray
- `value`: the value of the elements
- `l`: the layout of the pyarray

**pyarray** (`const` `shape_type` *&shape*, `const` `strides_type` *&strides*, `const_reference` *value*)

Allocates an uninitialized pyarray with the specified shape and strides.

Elements are initialized to the specified value.

**Parameters**

- `shape`: the shape of the pyarray
- `strides`: the strides of the pyarray
- `value`: the value of the elements

**pyarray** (`const` `shape_type` *&shape*, `const` `strides_type` *&strides*)

Allocates an uninitialized pyarray with the specified shape and strides.

**Parameters**

- `shape`: the shape of the pyarray
- `strides`: the strides of the pyarray

**template** `<class S>`

`pyarray``<T, L>` **from\_shape** (`S` *&&shape*)

Allocates and returns an pyarray with the specified shape.

**Parameters**

- `shape`: the shape of the pyarray

**Copy semantic**

**pyarray** (`const` `self_type` *&rhs*)

The copy constructor.

`auto` **operator=** (`const` `self_type` *&rhs*)

The assignment operator.

### Extended copy semantic

```
template <class E>
pyarray (const xexpression<E> &e)
    The extended copy constructor.
```

```
template <class E>
auto operator= (const xexpression<E> &e)
    The extended assignment operator.
```

### pytensor

```
template <class T, std::size_t N, layout_type L>
class pytensor : public xt::pycontainer<pytensor<T, N, L>>, public xcontainer_semantic<pytensor<T, N, L>>
    Multidimensional container providing the xtensor container semantics wrapping a numpy array.
```

pytensor is similar to the xtensor container in that it has a static dimensionality.

Unlike with the pyarray container, pytensor cannot be reshaped with a different number of dimensions and reshapes are not reflected on the Python side. However, pytensor has benefits compared to pyarray in terms of performances. pytensor shapes are stack-allocated which makes iteration upon pytensor faster than with pyarray.

See [pyarray](#)

#### Template Parameters

- T: The type of the element stored in the pyarray.

### Constructors

```
pytensor ()
    Allocates an uninitialized pytensor that holds 1 element.
```

```
pytensor (nested_initializer_list_t<T, N> t)
    Allocates a pytensor with a nested initializer list.
```

```
pytensor (const shape_type &shape, layout_type l = layout_type::row_major)
    Allocates an uninitialized pytensor with the specified shape and layout.
```

#### Parameters

- shape: the shape of the pytensor
- l: the layout\_type of the pytensor

```
pytensor (const shape_type &shape, const_reference value, layout_type l = lay-
    out_type::row_major)
    Allocates a pytensor with the specified shape and layout.
```

Elements are initialized to the specified value.

#### Parameters

- shape: the shape of the pytensor
- value: the value of the elements
- l: the layout\_type of the pytensor



**pytensor** (**const** shape\_type &shape, **const** strides\_type &strides, const\_reference value)

Allocates an uninitialized pytensor with the specified shape and strides.

Elements are initialized to the specified value.

#### Parameters

- shape: the shape of the pytensor
- strides: the strides of the pytensor
- value: the value of the elements

**pytensor** (**const** shape\_type &shape, **const** strides\_type &strides)

Allocates an uninitialized pytensor with the specified shape and strides.

#### Parameters

- shape: the shape of the pytensor
- strides: the strides of the pytensor

**template** <class S>

*pytensor*<T, N, L> **from\_shape** (S &&shape)

Allocates and returns an pytensor with the specified shape.

#### Parameters

- shape: the shape of the pytensor

### Copy semantic

**pytensor** (**const** self\_type &rhs)

The copy constructor.

auto **operator=** (**const** self\_type &rhs)

The assignment operator.

### Extended copy semantic

**template** <class E>

**pytensor** (**const** xexpression<E> &e)

The extended copy constructor.

**template** <class E>

auto **operator=** (**const** xexpression<E> &e)

The extended assignment operator.

## 2.6.2 Numpy universal functions

### pyvectorize

**template** <class R, class... Args>

pyvectorizer<R (\*)(Args...), R, Args...> xt : **pyvectorize**

R (\*f)Args...Create numpy universal function from scalar function.

## 2.7 Releasing xtensor-python

### 2.7.1 Releasing a new version

From the master branch of xtensor-python

- Make sure that you are in sync with the master branch of the upstream remote.
- In file `xtensor_python_config.hpp`, set the macros for `XTENSOR_PYTHON_VERSION_MAJOR`, `XTENSOR_PYTHON_VERSION_MINOR` and `XTENSOR_PYTHON_VERSION_PATCH` to the desired values.
- Update the readme file w.r.t. dependencies on `xtensor` and `pybind11`.
- Stage the changes (`git add`), commit the changes (`git commit`) and add a tag of the form `Major.minor.patch`. It is important to not add any other content to the tag name.
- Push the new commit and tag to the main repository. (`git push`, and `git push --tags`)

### 2.7.2 Updating the conda-forge recipe

xtensor-python has been packaged for the conda package manager. Once the new tag has been pushed on GitHub, edit the conda-forge recipe for xtensor in the following fashion:

- Update the version number to the new `Major.minor.patch`.
- Set the build number to 0.
- Update the hash of the source tarball.
- Check for the versions of the dependencies.
- Optionally, rerender the conda-forge feedstock.

## X

`xt::pyarray` (*C++ class*), 10  
`xt::pyarray::from_shape` (*C++ function*), 11  
`xt::pyarray::operator=` (*C++ function*), 11, 12  
`xt::pyarray::pyarray` (*C++ function*), 10–12  
`xt::pytensor` (*C++ class*), 12  
`xt::pytensor::from_shape` (*C++ function*), 13  
`xt::pytensor::operator=` (*C++ function*), 13  
`xt::pytensor::pytensor` (*C++ function*), 12, 13  
`xt::pyvectorize` (*C++ function*), 13