

---

# **npctypes Documentation**

***Release 0.0.3+0.g02b059b.dirty***

**John Kirkham**

**Nov 12, 2017**



---

## Contents

---

<b>1</b>	<b>npctypes</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Usage</b>	<b>7</b>
<b>4</b>	<b>API</b>	<b>9</b>
<b>5</b>	<b>Contributing</b>	<b>13</b>
<b>6</b>	<b>Indices and tables</b>	<b>17</b>
	<b>Python Module Index</b>	<b>19</b>



Contents:



A Python package for working with NumPy arrays and ctypes arrays.

- Free software: BSD 3-Clause
- Documentation: <https://npctypes.readthedocs.io>.

## 1.1 Features

- TODO

## 1.2 Credits

This package was created with [Cookiecutter](#) and the [nanshe-org/nanshe-cookiecutter](#) project template.





### 2.1 Stable release

To install npctypes, run this command in your terminal:

```
$ pip install npctypes
```

This is the preferred method to install npctypes, as it will always install the most recent stable release.

If you don't have [pip](#) installed, this [Python installation guide](#) can guide you through the process.

### 2.2 From sources

The sources for npctypes can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
$ git clone git://github.com/jakirkham/npctypes
```

Or download the [tarball](#):

```
$ curl -OL https://github.com/jakirkham/npctypes/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```



## CHAPTER 3

---

### Usage

---

To use npctypes in a project:

```
import npctypes
```



## 4.1 npctypes package

### 4.1.1 Submodules

#### npctypes.shared module

`npctypes.shared.as_ndarray(*args, **kws)`

Context manager to provide NumPy ndarray views of NDArray instances.

##### Parameters

- **shape** (*tuple of ints*) – Shape of the array to allocate.
- **dtype** (*type*) – Type of the array to allocate.
- **order** (*char or None*) – Order of the array ('C', 'F', or None). Defaults to None.

##### Returns

Custom Array instance allocated on the shared process heap.

**Return type** `ctypes.Array`

#### Examples

```
>>> a = ndarray((2,3), float)
>>> with as_ndarray(a) as nd_a:
...     nd_a[...] = 0
...     print(nd_a)
[[ 0.  0.  0.]
 [ 0.  0.  0.]
```

```
>>> a = ndarray((2,3), float)
>>> with as_ndarray(a) as nd_a:
...     for i in range(nd_a.size):
...         nd_a.flat[i] = i
...
...     print(nd_a)
[[ 0.  1.  2.]
 [ 3.  4.  5.]]
```

`npctypes.shared.ndarray` (*shape, dtype, order=None*)

Factory to generate N-D Arrays shared across process boundaries.

This creates a custom dynamic type (if one doesn't already exist) that is a `ctypes.Array` instance. If one does already exist, we reuse it so that things like type comparisons work. In addition to the typical properties that “`ctypes.Array`”s have, this tracks its number of dimensions, shape, and order ('C' or 'F' for C and Fortran respectively). Having this information allows us to easily construct a NumPy `ndarray` in other processes.

#### Parameters

- **shape** (*tuple of ints*) – Shape of the array to allocate.
- **dtype** (*type*) – Type of the array to allocate.
- **order** (*char or None*) – Order of the array ('C', 'F', or None). Defaults to None.

#### Returns

**Custom Array (NDArray) instance** allocated on the shared process heap.

**Return type** `ctypes.Array`

### Examples

```
>>> ndarray((2,3), float)
<npctypes.shared.NDArray_<f8_2d_2x3_C object at 0x...>
```

```
>>> ndarray((2,3), float, order='F')
<npctypes.shared.NDArray_<f8_2d_2x3_F object at 0x...>
```

## npctypes.types module

`npctypes.types.ctype` (*a\_type*)

Takes a `numpy.dtype` or any type that can be converted to a `numpy.dtype` and returns its equivalent `ctype`.

**Parameters** **a\_type** (*type*) – the type to find an equivalent `ctype` to.

**Returns** the `ctype` equivalent to the `dtype` provided.

**Return type** (*ctype*)

### Examples

```
>>> ctype(float)
<class 'ctypes.c_double'>
```

```
>>> ctype(numpy.float64)
<class 'ctypes.c_double'>
```

```
>>> ctype(numpy.float32)
<class 'ctypes.c_float'>
```

```
>>> ctype(numpy.dtype(numpy.float32))
<class 'ctypes.c_float'>
```

```
>>> ctype(int)
<class 'ctypes.c_long'>
```

`npctypes.types.get_ndpointer_type(a)`

Takes a `numpy.ndarray` and gets a pointer type for that array.

**Parameters** `a` (`ndarray`) – the ndarray to get the pointer type for.

**Returns** the pointer type associated with this array.

**Return type** (`PyCSimpleType`)

## Examples

```
>>> a = numpy.zeros((3, 4), dtype=float)
>>> a_ptr = get_ndpointer_type(a)
```

```
>>> a_ptr
<class 'numpy.ctypeslib.ndpointer_<f8_2d_3x4_C_CONTIGUOUS_ALIGNED_WRITEABLE_
↳OWNDATA'>
```

```
>>> a_ptr._dtype_
dtype('float64')
>>> a_ptr._ndim_
2
>>> a_ptr._shape_
(3, 4)
>>> a_ptr._flags_
1285
>>> numpy.ctypeslib.flagsobj(a_ptr._flags_)
C_CONTIGUOUS : True
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : False
ALIGNED : True
UPDATEIFCOPY : False
```

`npctypes.types.tinfo(a_type)`

Takes a `numpy.dtype` or any type that can be converted to a `numpy.dtype` and returns its info.

**Parameters** `a_type` (`type`) – the type to find info for.

**Returns** info about the type.

**Return type** (`np.core.getlimits.info`)

## Examples

```
>>> tinfo(float)
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308,
↪ dtype=float64)
```

```
>>> tinfo(numpy.float64)
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308,
↪ dtype=float64)
```

```
>>> tinfo(numpy.float32)
finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

```
>>> tinfo(complex)
finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308,
↪ dtype=float64)
```

```
>>> tinfo(int)
iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```



Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

## 5.1 Types of Contributions

### 5.1.1 Report Bugs

Report bugs at <https://github.com/jakirkham/npctypes/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

### 5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

### 5.1.4 Write Documentation

npctypes could always use more documentation, whether as part of the official npctypes docs, in docstrings, or even on the web in blog posts, articles, and such.

### 5.1.5 Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/jakirkham/npctypes/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 5.2 Get Started!

Ready to contribute? Here's how to set up *npctypes* for local development.

1. Fork the *npctypes* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/npctypes.git
```

3. Install your local copy into an environment. Assuming you have conda installed, this is how you set up your fork for local development (on Windows drop *source*). Replace “<some version>” with the Python version used for testing.:

```
$ conda create -n npctypesenv python="<some version>"
$ source activate npctypesenv
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions:

```
$ flake8 npctypes tests
$ python setup.py test or py.test
```

To get flake8, just conda install it into your environment.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

## 5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5, and 3.6. Check [https://travis-ci.org/jakirkham/npctypes/pull\\_requests](https://travis-ci.org/jakirkham/npctypes/pull_requests) and make sure that the tests pass for all supported Python versions.

## 5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_npctypes
```



## CHAPTER 6

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### n

`npctypes`, [9](#)  
`npctypes.shared`, [9](#)  
`npctypes.types`, [10](#)





## A

`as_ndarray()` (in module `npctypes.shared`), 9

## C

`ctype()` (in module `npctypes.types`), 10

## G

`get_ndpointer_type()` (in module `npctypes.types`), 11

## N

`ndarray()` (in module `npctypes.shared`), 10

`npctypes` (module), 9

`npctypes.shared` (module), 9

`npctypes.types` (module), 10

## T

`tinfo()` (in module `npctypes.types`), 11