Search the docs ...

What is NumPy?
Installation
NumPy quickstart
NumPy: the absolute basics for beginners
NumPy fundamentals
Miscellaneous
NumPy for MATLAB users
Building from source
Using NumPy C-API
NumPy Tutorials
NumPy How Tos
For downstream package authors
F2PY user guide and reference manual
Glossary
Under-the-hood Documentation for developers
Reporting bugs
Release notes
NumPy license

Miscellaneous

IEEE 754 Floating Point Special Values

Special values defined in numpy: nan, inf,

NaNs can be used as a poor-man's mask (if you don't care what the original value was)

Note: cannot use equality to test NaNs. E.g.:

```
>>> myarr = np.array([1., 0., np.nan, 3.])
>>> np.nonzero(myarr == np.nan)
(array([], dtype=int64),)
>>> np.nan == np.nan  # is always False! Use special numpy functions instead.
False
>>> myarr[myarr == np.nan] = 0. # doesn't work
>>> myarr
array([  1.,   0.,  nan,   3.])
>>> myarr[np.isnan(myarr)] = 0. # use this instead find
>>> myarr
array([1.,  0.,  0.,  3.])
```

Other related special value functions:

```
isinf():    True if value is inf
isfinite(): True if not nan or inf
nan_to_num(): Map nan to 0, inf to max float, -inf to min float
```

The following corresponds to the usual functions except that nans are excluded from the results:

```
nansum()
nanmax()
nanmin()
nanargmax()
nanargmin()

>>> x = np.arange(10.)
>>> x[3] = np.nan
>>> x.sum()
nan
>>> np.nansum(x)
42.0
```

How numpy handles numerical exceptions

The default is to `'warn'` for `invalid`, `divide`, and `overflow` and `'ignore'` for `underflow`. But this can be changed, and it can be set individually for different kinds of exceptions. The different behaviors are:

- 'ignore' : Take no action when the exception occurs.
- 'warn' : Print a *RuntimeWarning* (via the Python **warnings** module).
- 'raise' : Raise a *FloatingPointError*.
- 'call' : Call a function specified using the *seterrcall* function.
- 'print' : Print a warning directly to `stdout`.
- 'log' : Record error in a Log object specified by *seterrcall*.

These behaviors can be set for all kinds of errors or specific ones:

- all : apply to all numeric exceptions
- invalid : when NaNs are generated
- divide : divide by zero (for integers as well!)
- overflow : floating point overflows
- underflow : floating point underflows

Note that integer divide-by-zero is handled by the same machinery. These behaviors are set on a per-thread basis.

Examples

```
>>> oldsettings = np.seterr(all='warn')
>>> np.zeros(5,dtype=np.float32)/0.
Traceback (most recent call last):
...
RuntimeWarning: invalid value encountered in divide
>>> j = np.seterr(under='ignore')
>>> np.array([1.e-100])**10
array([0.])
>>> j = np.seterr(invalid='raise')
>>> np.sqrt(np.array([-1.]))
Traceback (most recent call last):
...
FloatingPointError: invalid value encountered in sqrt
>>> def errorhandler(errstr, errflag):
...      print("saw stupid error!")
>>> np.seterrcall(errorhandler)
>>> j = np.seterr(all='call')
>>> np.zeros(5, dtype=np.int32)/0
saw stupid error!
array([nan, nan, nan, nan, nan])
>>> j = np.seterr(**oldsettings) # restore previous
...                              # error-handling settings
```

Interfacing to C

Only a survey of the choices. Little detail on how each works.

1. Bare metal, wrap your own C-code manually.

- Plusses:
  - Efficient
  - No dependencies on other tools
- Minuses:
  - Lots of learning overhead:
    - need to learn basics of Python C API
    - need to learn basics of numpy C API
    - need to learn how to handle reference counting and love it.
  - Reference counting often difficult to get right.
    - getting it wrong leads to memory leaks, and worse, segfaults

2. Cython

- Plusses:
  - avoid learning C API’s
  - no dealing with reference counting
  - can code in pseudo python and generate C code
  - can also interface to existing C code
  - should shield you from changes to Python C api
  - has become the de-facto standard within the scientific Python community
  - fast indexing support for arrays
- Minuses:
  - Can write code in non-standard form which may become obsolete
  - Not as flexible as manual wrapping

3. ctypes

- Plusses:
  - part of Python standard library
  - good for interfacing to existing shareable libraries, particularly Windows DLLs
  - avoids API/reference counting issues
  - good numpy support: arrays have all these in their ctypes attribute:

```
a.ctypes.data
a.ctypes.data_as
a.ctypes.shape
a.ctypes.shape_as
a.ctypes.strides
a.ctypes.strides_as
```

- Minuses:
  - can’t use for writing code to be turned into C extensions, only a wrapper tool.

4. SWIG (automatic wrapper generator)

- Plusses:
  - around a long time
  - multiple scripting language support
  - C++ support
  - Good for wrapping large (many functions) existing C libraries
- Minuses:
  - generates lots of code between Python and the C code
  - can cause performance problems that are nearly impossible to optimize out
  - interface files can be hard to write
  - doesn’t necessarily avoid reference counting issues or needing to know API’s

5. Psyco

- Plusses:
  - Turns pure python into efficient machine code through jit-like optimizations
  - very fast when it optimizes well
- Minuses:
  - Only on intel (windows?)
  - Doesn’t do much for numpy?

Interfacing to Fortran:

The clear choice to wrap Fortran code is f2py.

Pyfort is an older alternative, but not supported any longer. Fwrap is a newer project that looked promising but isn't being developed any longer.

Interfacing to C++:

1. Cython
2. CXX
3. Boost.python
4. SWIG
5. SIP (used mainly in PyQT)

Previous
Interoperability with NumPy

Next
NumPy for MATLAB users