===========

NumPy C-API

===========

::

unsigned int

PyArray\_GetNDArrayCVersion(void )

Included at the very first so not auto-grabbed and thus not labeled.

::

int

PyArray\_SetNumericOps(PyObject \*dict)

Set internal structure with number functions that all arrays will use

::

PyObject \*

PyArray\_GetNumericOps(void )

Get dictionary showing number functions that all arrays will use

::

int

PyArray\_INCREF(PyArrayObject \*mp)

For object arrays, increment all internal references.

::

int

PyArray\_XDECREF(PyArrayObject \*mp)

Decrement all internal references for object arrays.

(or arrays with object fields)

::

void

PyArray\_SetStringFunction(PyObject \*op, int repr)

Set the array print function to be a Python function.

::

PyArray\_Descr \*

PyArray\_DescrFromType(int type)

Get the PyArray\_Descr structure for a type.

::

PyObject \*

PyArray\_TypeObjectFromType(int type)

Get a typeobject from a type-number -- can return NULL.

New reference

::

char \*

PyArray\_Zero(PyArrayObject \*arr)

Get pointer to zero of correct type for array.

::

char \*

PyArray\_One(PyArrayObject \*arr)

Get pointer to one of correct type for array

::

PyObject \*

PyArray\_CastToType(PyArrayObject \*arr, PyArray\_Descr \*dtype, int

is\_f\_order)

For backward compatibility

Cast an array using typecode structure.

steals reference to dtype --- cannot be NULL

This function always makes a copy of arr, even if the dtype

doesn't change.

::

int

PyArray\_CastTo(PyArrayObject \*out, PyArrayObject \*mp)

Cast to an already created array.

::

int

PyArray\_CastAnyTo(PyArrayObject \*out, PyArrayObject \*mp)

Cast to an already created array. Arrays don't have to be "broadcastable"

Only requirement is they have the same number of elements.

::

int

PyArray\_CanCastSafely(int fromtype, int totype)

Check the type coercion rules.

::

npy\_bool

PyArray\_CanCastTo(PyArray\_Descr \*from, PyArray\_Descr \*to)

leaves reference count alone --- cannot be NULL

PyArray\_CanCastTypeTo is equivalent to this, but adds a 'casting'

parameter.

::

int

PyArray\_ObjectType(PyObject \*op, int minimum\_type)

Return the typecode of the array a Python object would be converted to

Returns the type number the result should have, or NPY\_NOTYPE on error.

::

PyArray\_Descr \*

PyArray\_DescrFromObject(PyObject \*op, PyArray\_Descr \*mintype)

new reference -- accepts NULL for mintype

::

PyArrayObject \*\*

PyArray\_ConvertToCommonType(PyObject \*op, int \*retn)

::

PyArray\_Descr \*

PyArray\_DescrFromScalar(PyObject \*sc)

Return descr object from array scalar.

New reference

::

PyArray\_Descr \*

PyArray\_DescrFromTypeObject(PyObject \*type)

::

npy\_intp

PyArray\_Size(PyObject \*op)

Compute the size of an array (in number of items)

::

PyObject \*

PyArray\_Scalar(void \*data, PyArray\_Descr \*descr, PyObject \*base)

Get scalar-equivalent to a region of memory described by a descriptor.

::

PyObject \*

PyArray\_FromScalar(PyObject \*scalar, PyArray\_Descr \*outcode)

Get 0-dim array from scalar

0-dim array from array-scalar object

always contains a copy of the data

unless outcode is NULL, it is of void type and the referrer does

not own it either.

steals reference to outcode

::

void

PyArray\_ScalarAsCtype(PyObject \*scalar, void \*ctypeptr)

Convert to c-type

no error checking is performed -- ctypeptr must be same type as scalar

in case of flexible type, the data is not copied

into ctypeptr which is expected to be a pointer to pointer

::

int

PyArray\_CastScalarToCtype(PyObject \*scalar, void

\*ctypeptr, PyArray\_Descr \*outcode)

Cast Scalar to c-type

The output buffer must be large-enough to receive the value

Even for flexible types which is different from ScalarAsCtype

where only a reference for flexible types is returned

This may not work right on narrow builds for NumPy unicode scalars.

::

int

PyArray\_CastScalarDirect(PyObject \*scalar, PyArray\_Descr

\*indescr, void \*ctypeptr, int outtype)

Cast Scalar to c-type

::

PyObject \*

PyArray\_ScalarFromObject(PyObject \*object)

Get an Array Scalar From a Python Object

Returns NULL if unsuccessful but error is only set if another error occurred.

Currently only Numeric-like object supported.

::

PyArray\_VectorUnaryFunc \*

PyArray\_GetCastFunc(PyArray\_Descr \*descr, int type\_num)

Get a cast function to cast from the input descriptor to the

output type\_number (must be a registered data-type).

Returns NULL if un-successful.

::

PyObject \*

PyArray\_FromDims(int NPY\_UNUSED(nd) , int \*NPY\_UNUSED(d) , int

NPY\_UNUSED(type) )

Deprecated, use PyArray\_SimpleNew instead.

::

PyObject \*

PyArray\_FromDimsAndDataAndDescr(int NPY\_UNUSED(nd) , int

\*NPY\_UNUSED(d) , PyArray\_Descr

\*descr, char \*NPY\_UNUSED(data) )

Deprecated, use PyArray\_NewFromDescr instead.

::

PyObject \*

PyArray\_FromAny(PyObject \*op, PyArray\_Descr \*newtype, int

min\_depth, int max\_depth, int flags, PyObject

\*context)

Does not check for NPY\_ARRAY\_ENSURECOPY and NPY\_ARRAY\_NOTSWAPPED in flags

Steals a reference to newtype --- which can be NULL

::

PyObject \*

PyArray\_EnsureArray(PyObject \*op)

This is a quick wrapper around

PyArray\_FromAny(op, NULL, 0, 0, NPY\_ARRAY\_ENSUREARRAY, NULL)

that special cases Arrays and PyArray\_Scalars up front

It \*steals a reference\* to the object

It also guarantees that the result is PyArray\_Type

Because it decrefs op if any conversion needs to take place

so it can be used like PyArray\_EnsureArray(some\_function(...))

::

PyObject \*

PyArray\_EnsureAnyArray(PyObject \*op)

::

PyObject \*

PyArray\_FromFile(FILE \*fp, PyArray\_Descr \*dtype, npy\_intp num, char

\*sep)

Given a ``FILE \*`` pointer ``fp``, and a ``PyArray\_Descr``, return an

array corresponding to the data encoded in that file.

The reference to `dtype` is stolen (it is possible that the passed in

dtype is not held on to).

The number of elements to read is given as ``num``; if it is < 0, then

then as many as possible are read.

If ``sep`` is NULL or empty, then binary data is assumed, else

text data, with ``sep`` as the separator between elements. Whitespace in

the separator matches any length of whitespace in the text, and a match

for whitespace around the separator is added.

For memory-mapped files, use the buffer interface. No more data than

necessary is read by this routine.

::

PyObject \*

PyArray\_FromString(char \*data, npy\_intp slen, PyArray\_Descr

\*dtype, npy\_intp num, char \*sep)

Given a pointer to a string ``data``, a string length ``slen``, and

a ``PyArray\_Descr``, return an array corresponding to the data

encoded in that string.

If the dtype is NULL, the default array type is used (double).

If non-null, the reference is stolen.

If ``slen`` is < 0, then the end of string is used for text data.

It is an error for ``slen`` to be < 0 for binary data (since embedded NULLs

would be the norm).

The number of elements to read is given as ``num``; if it is < 0, then

then as many as possible are read.

If ``sep`` is NULL or empty, then binary data is assumed, else

text data, with ``sep`` as the separator between elements. Whitespace in

the separator matches any length of whitespace in the text, and a match

for whitespace around the separator is added.

::

PyObject \*

PyArray\_FromBuffer(PyObject \*buf, PyArray\_Descr \*type, npy\_intp

count, npy\_intp offset)

::

PyObject \*

PyArray\_FromIter(PyObject \*obj, PyArray\_Descr \*dtype, npy\_intp count)

steals a reference to dtype (which cannot be NULL)

::

PyObject \*

PyArray\_Return(PyArrayObject \*mp)

Return either an array or the appropriate Python object if the array

is 0d and matches a Python type.

steals reference to mp

::

PyObject \*

PyArray\_GetField(PyArrayObject \*self, PyArray\_Descr \*typed, int

offset)

Get a subset of bytes from each element of the array

steals reference to typed, must not be NULL

::

int

PyArray\_SetField(PyArrayObject \*self, PyArray\_Descr \*dtype, int

offset, PyObject \*val)

Set a subset of bytes from each element of the array

steals reference to dtype, must not be NULL

::

PyObject \*

PyArray\_Byteswap(PyArrayObject \*self, npy\_bool inplace)

::

PyObject \*

PyArray\_Resize(PyArrayObject \*self, PyArray\_Dims \*newshape, int

refcheck, NPY\_ORDER NPY\_UNUSED(order) )

Resize (reallocate data). Only works if nothing else is referencing this

array and it is contiguous. If refcheck is 0, then the reference count is

not checked and assumed to be 1. You still must own this data and have no

weak-references and no base object.

::

int

PyArray\_MoveInto(PyArrayObject \*dst, PyArrayObject \*src)

Move the memory of one array into another, allowing for overlapping data.

Returns 0 on success, negative on failure.

::

int

PyArray\_CopyInto(PyArrayObject \*dst, PyArrayObject \*src)

Copy an Array into another array.

Broadcast to the destination shape if necessary.

Returns 0 on success, -1 on failure.

::

int

PyArray\_CopyAnyInto(PyArrayObject \*dst, PyArrayObject \*src)

Copy an Array into another array -- memory must not overlap

Does not require src and dest to have "broadcastable" shapes

(only the same number of elements).

TODO: For NumPy 2.0, this could accept an order parameter which

only allows NPY\_CORDER and NPY\_FORDER. Could also rename

this to CopyAsFlat to make the name more intuitive.

Returns 0 on success, -1 on error.

::

int

PyArray\_CopyObject(PyArrayObject \*dest, PyObject \*src\_object)

::

PyObject \*

PyArray\_NewCopy(PyArrayObject \*obj, NPY\_ORDER order)

Copy an array.

::

PyObject \*

PyArray\_ToList(PyArrayObject \*self)

To List

::

PyObject \*

PyArray\_ToString(PyArrayObject \*self, NPY\_ORDER order)

::

int

PyArray\_ToFile(PyArrayObject \*self, FILE \*fp, char \*sep, char \*format)

To File

::

int

PyArray\_Dump(PyObject \*self, PyObject \*file, int protocol)

::

PyObject \*

PyArray\_Dumps(PyObject \*self, int protocol)

::

int

PyArray\_ValidType(int type)

Is the typenum valid?

::

void

PyArray\_UpdateFlags(PyArrayObject \*ret, int flagmask)

Update Several Flags at once.

::

PyObject \*

PyArray\_New(PyTypeObject \*subtype, int nd, npy\_intp const \*dims, int

type\_num, npy\_intp const \*strides, void \*data, int

itemsize, int flags, PyObject \*obj)

Generic new array creation routine.

::

PyObject \*

PyArray\_NewFromDescr(PyTypeObject \*subtype, PyArray\_Descr \*descr, int

nd, npy\_intp const \*dims, npy\_intp const

\*strides, void \*data, int flags, PyObject \*obj)

Generic new array creation routine.

steals a reference to descr. On failure or when dtype->subarray is

true, dtype will be decrefed.

::

PyArray\_Descr \*

PyArray\_DescrNew(PyArray\_Descr \*base)

base cannot be NULL

::

PyArray\_Descr \*

PyArray\_DescrNewFromType(int type\_num)

::

double

PyArray\_GetPriority(PyObject \*obj, double default\_)

Get Priority from object

::

PyObject \*

PyArray\_IterNew(PyObject \*obj)

Get Iterator.

::

PyObject\*

PyArray\_MultiIterNew(int n, ... )

Get MultiIterator,

::

int

PyArray\_PyIntAsInt(PyObject \*o)

::

npy\_intp

PyArray\_PyIntAsIntp(PyObject \*o)

::

int

PyArray\_Broadcast(PyArrayMultiIterObject \*mit)

::

void

PyArray\_FillObjectArray(PyArrayObject \*arr, PyObject \*obj)

Assumes contiguous

::

int

PyArray\_FillWithScalar(PyArrayObject \*arr, PyObject \*obj)

::

npy\_bool

PyArray\_CheckStrides(int elsize, int nd, npy\_intp numbytes, npy\_intp

offset, npy\_intp \*dims, npy\_intp \*newstrides)

::

PyArray\_Descr \*

PyArray\_DescrNewByteorder(PyArray\_Descr \*self, char newendian)

returns a copy of the PyArray\_Descr structure with the byteorder

altered:

no arguments: The byteorder is swapped (in all subfields as well)

single argument: The byteorder is forced to the given state

(in all subfields as well)

Valid states: ('big', '>') or ('little' or '<')

('native', or '=')

If a descr structure with | is encountered it's own

byte-order is not changed but any fields are:

Deep bytorder change of a data-type descriptor

Leaves reference count of self unchanged --- does not DECREF self \*\*\*

::

PyObject \*

PyArray\_IterAllButAxis(PyObject \*obj, int \*inaxis)

Get Iterator that iterates over all but one axis (don't use this with

PyArray\_ITER\_GOTO1D). The axis will be over-written if negative

with the axis having the smallest stride.

::

PyObject \*

PyArray\_CheckFromAny(PyObject \*op, PyArray\_Descr \*descr, int

min\_depth, int max\_depth, int requires, PyObject

\*context)

steals a reference to descr -- accepts NULL

::

PyObject \*

PyArray\_FromArray(PyArrayObject \*arr, PyArray\_Descr \*newtype, int

flags)

steals reference to newtype --- acc. NULL

::

PyObject \*

PyArray\_FromInterface(PyObject \*origin)

::

PyObject \*

PyArray\_FromStructInterface(PyObject \*input)

::

PyObject \*

PyArray\_FromArrayAttr(PyObject \*op, PyArray\_Descr \*typecode, PyObject

\*context)

::

NPY\_SCALARKIND

PyArray\_ScalarKind(int typenum, PyArrayObject \*\*arr)

ScalarKind

Returns the scalar kind of a type number, with an

optional tweak based on the scalar value itself.

If no scalar is provided, it returns INTPOS\_SCALAR

for both signed and unsigned integers, otherwise

it checks the sign of any signed integer to choose

INTNEG\_SCALAR when appropriate.

::

int

PyArray\_CanCoerceScalar(int thistype, int neededtype, NPY\_SCALARKIND

scalar)

Determines whether the data type 'thistype', with

scalar kind 'scalar', can be coerced into 'neededtype'.

::

PyObject \*

PyArray\_NewFlagsObject(PyObject \*obj)

Get New ArrayFlagsObject

::

npy\_bool

PyArray\_CanCastScalar(PyTypeObject \*from, PyTypeObject \*to)

See if array scalars can be cast.

TODO: For NumPy 2.0, add a NPY\_CASTING parameter.

::

int

PyArray\_CompareUCS4(npy\_ucs4 \*s1, npy\_ucs4 \*s2, size\_t len)

::

int

PyArray\_RemoveSmallest(PyArrayMultiIterObject \*multi)

Adjusts previously broadcasted iterators so that the axis with

the smallest sum of iterator strides is not iterated over.

Returns dimension which is smallest in the range [0,multi->nd).

A -1 is returned if multi->nd == 0.

don't use with PyArray\_ITER\_GOTO1D because factors are not adjusted

::

int

PyArray\_ElementStrides(PyObject \*obj)

::

void

PyArray\_Item\_INCREF(char \*data, PyArray\_Descr \*descr)

XINCREF all objects in a single array item. This is complicated for

structured datatypes where the position of objects needs to be extracted.

The function is execute recursively for each nested field or subarrays dtype

such as as `np.dtype([("field1", "O"), ("field2", "f,O", (3,2))])`

::

void

PyArray\_Item\_XDECREF(char \*data, PyArray\_Descr \*descr)

XDECREF all objects in a single array item. This is complicated for

structured datatypes where the position of objects needs to be extracted.

The function is execute recursively for each nested field or subarrays dtype

such as as `np.dtype([("field1", "O"), ("field2", "f,O", (3,2))])`

::

PyObject \*

PyArray\_FieldNames(PyObject \*fields)

Return the tuple of ordered field names from a dictionary.

::

PyObject \*

PyArray\_Transpose(PyArrayObject \*ap, PyArray\_Dims \*permute)

Return Transpose.

::

PyObject \*

PyArray\_TakeFrom(PyArrayObject \*self0, PyObject \*indices0, int

axis, PyArrayObject \*out, NPY\_CLIPMODE clipmode)

Take

::

PyObject \*

PyArray\_PutTo(PyArrayObject \*self, PyObject\*values0, PyObject

\*indices0, NPY\_CLIPMODE clipmode)

Put values into an array

::

PyObject \*

PyArray\_PutMask(PyArrayObject \*self, PyObject\*values0, PyObject\*mask0)

Put values into an array according to a mask.

::

PyObject \*

PyArray\_Repeat(PyArrayObject \*aop, PyObject \*op, int axis)

Repeat the array.

::

PyObject \*

PyArray\_Choose(PyArrayObject \*ip, PyObject \*op, PyArrayObject

\*out, NPY\_CLIPMODE clipmode)

::

int

PyArray\_Sort(PyArrayObject \*op, int axis, NPY\_SORTKIND which)

Sort an array in-place

::

PyObject \*

PyArray\_ArgSort(PyArrayObject \*op, int axis, NPY\_SORTKIND which)

ArgSort an array

::

PyObject \*

PyArray\_SearchSorted(PyArrayObject \*op1, PyObject \*op2, NPY\_SEARCHSIDE

side, PyObject \*perm)

Search the sorted array op1 for the location of the items in op2. The

result is an array of indexes, one for each element in op2, such that if

the item were to be inserted in op1 just before that index the array

would still be in sorted order.

Parameters

----------

op1 : PyArrayObject \*

Array to be searched, must be 1-D.

op2 : PyObject \*

Array of items whose insertion indexes in op1 are wanted

side : {NPY\_SEARCHLEFT, NPY\_SEARCHRIGHT}

If NPY\_SEARCHLEFT, return first valid insertion indexes

If NPY\_SEARCHRIGHT, return last valid insertion indexes

perm : PyObject \*

Permutation array that sorts op1 (optional)

Returns

-------

ret : PyObject \*

New reference to npy\_intp array containing indexes where items in op2

could be validly inserted into op1. NULL on error.

Notes

-----

Binary search is used to find the indexes.

::

PyObject \*

PyArray\_ArgMax(PyArrayObject \*op, int axis, PyArrayObject \*out)

ArgMax

::

PyObject \*

PyArray\_ArgMin(PyArrayObject \*op, int axis, PyArrayObject \*out)

ArgMin

::

PyObject \*

PyArray\_Reshape(PyArrayObject \*self, PyObject \*shape)

Reshape

::

PyObject \*

PyArray\_Newshape(PyArrayObject \*self, PyArray\_Dims \*newdims, NPY\_ORDER

order)

New shape for an array

::

PyObject \*

PyArray\_Squeeze(PyArrayObject \*self)

return a new view of the array object with all of its unit-length

dimensions squeezed out if needed, otherwise

return the same array.

::

PyObject \*

PyArray\_View(PyArrayObject \*self, PyArray\_Descr \*type, PyTypeObject

\*pytype)

View

steals a reference to type -- accepts NULL

::

PyObject \*

PyArray\_SwapAxes(PyArrayObject \*ap, int a1, int a2)

SwapAxes

::

PyObject \*

PyArray\_Max(PyArrayObject \*ap, int axis, PyArrayObject \*out)

Max

::

PyObject \*

PyArray\_Min(PyArrayObject \*ap, int axis, PyArrayObject \*out)

Min

::

PyObject \*

PyArray\_Ptp(PyArrayObject \*ap, int axis, PyArrayObject \*out)

Ptp

::

PyObject \*

PyArray\_Mean(PyArrayObject \*self, int axis, int rtype, PyArrayObject

\*out)

Mean

::

PyObject \*

PyArray\_Trace(PyArrayObject \*self, int offset, int axis1, int

axis2, int rtype, PyArrayObject \*out)

Trace

::

PyObject \*

PyArray\_Diagonal(PyArrayObject \*self, int offset, int axis1, int

axis2)

Diagonal

In NumPy versions prior to 1.7, this function always returned a copy of

the diagonal array. In 1.7, the code has been updated to compute a view

onto 'self', but it still copies this array before returning, as well as

setting the internal WARN\_ON\_WRITE flag. In a future version, it will

simply return a view onto self.

::

PyObject \*

PyArray\_Clip(PyArrayObject \*self, PyObject \*min, PyObject

\*max, PyArrayObject \*out)

Clip

::

PyObject \*

PyArray\_Conjugate(PyArrayObject \*self, PyArrayObject \*out)

Conjugate

::

PyObject \*

PyArray\_Nonzero(PyArrayObject \*self)

Nonzero

TODO: In NumPy 2.0, should make the iteration order a parameter.

::

PyObject \*

PyArray\_Std(PyArrayObject \*self, int axis, int rtype, PyArrayObject

\*out, int variance)

Set variance to 1 to by-pass square-root calculation and return variance

Std

::

PyObject \*

PyArray\_Sum(PyArrayObject \*self, int axis, int rtype, PyArrayObject

\*out)

Sum

::

PyObject \*

PyArray\_CumSum(PyArrayObject \*self, int axis, int rtype, PyArrayObject

\*out)

CumSum

::

PyObject \*

PyArray\_Prod(PyArrayObject \*self, int axis, int rtype, PyArrayObject

\*out)

Prod

::

PyObject \*

PyArray\_CumProd(PyArrayObject \*self, int axis, int

rtype, PyArrayObject \*out)

CumProd

::

PyObject \*

PyArray\_All(PyArrayObject \*self, int axis, PyArrayObject \*out)

All

::

PyObject \*

PyArray\_Any(PyArrayObject \*self, int axis, PyArrayObject \*out)

Any

::

PyObject \*

PyArray\_Compress(PyArrayObject \*self, PyObject \*condition, int

axis, PyArrayObject \*out)

Compress

::

PyObject \*

PyArray\_Flatten(PyArrayObject \*a, NPY\_ORDER order)

Flatten

::

PyObject \*

PyArray\_Ravel(PyArrayObject \*arr, NPY\_ORDER order)

Ravel

Returns a contiguous array

::

npy\_intp

PyArray\_MultiplyList(npy\_intp const \*l1, int n)

Multiply a List

::

int

PyArray\_MultiplyIntList(int const \*l1, int n)

Multiply a List of ints

::

void \*

PyArray\_GetPtr(PyArrayObject \*obj, npy\_intp const\*ind)

Produce a pointer into array

::

int

PyArray\_CompareLists(npy\_intp const \*l1, npy\_intp const \*l2, int n)

Compare Lists

::

int

PyArray\_AsCArray(PyObject \*\*op, void \*ptr, npy\_intp \*dims, int

nd, PyArray\_Descr\*typedescr)

Simulate a C-array

steals a reference to typedescr -- can be NULL

::

int

PyArray\_As1D(PyObject \*\*NPY\_UNUSED(op) , char \*\*NPY\_UNUSED(ptr) , int

\*NPY\_UNUSED(d1) , int NPY\_UNUSED(typecode) )

Convert to a 1D C-array

::

int

PyArray\_As2D(PyObject \*\*NPY\_UNUSED(op) , char \*\*\*NPY\_UNUSED(ptr) , int

\*NPY\_UNUSED(d1) , int \*NPY\_UNUSED(d2) , int

NPY\_UNUSED(typecode) )

Convert to a 2D C-array

::

int

PyArray\_Free(PyObject \*op, void \*ptr)

Free pointers created if As2D is called

::

int

PyArray\_Converter(PyObject \*object, PyObject \*\*address)

Useful to pass as converter function for O& processing in PyArgs\_ParseTuple.

This conversion function can be used with the "O&" argument for

PyArg\_ParseTuple. It will immediately return an object of array type

or will convert to a NPY\_ARRAY\_CARRAY any other object.

If you use PyArray\_Converter, you must DECREF the array when finished

as you get a new reference to it.

::

int

PyArray\_IntpFromSequence(PyObject \*seq, npy\_intp \*vals, int maxvals)

PyArray\_IntpFromSequence

Returns the number of integers converted or -1 if an error occurred.

vals must be large enough to hold maxvals

::

PyObject \*

PyArray\_Concatenate(PyObject \*op, int axis)

Concatenate

Concatenate an arbitrary Python sequence into an array.

op is a python object supporting the sequence interface.

Its elements will be concatenated together to form a single

multidimensional array. If axis is NPY\_MAXDIMS or bigger, then

each sequence object will be flattened before concatenation

::

PyObject \*

PyArray\_InnerProduct(PyObject \*op1, PyObject \*op2)

Numeric.innerproduct(a,v)

::

PyObject \*

PyArray\_MatrixProduct(PyObject \*op1, PyObject \*op2)

Numeric.matrixproduct(a,v)

just like inner product but does the swapaxes stuff on the fly

::

PyObject \*

PyArray\_CopyAndTranspose(PyObject \*op)

Copy and Transpose

Could deprecate this function, as there isn't a speed benefit over

calling Transpose and then Copy.

::

PyObject \*

PyArray\_Correlate(PyObject \*op1, PyObject \*op2, int mode)

Numeric.correlate(a1,a2,mode)

::

int

PyArray\_TypestrConvert(int itemsize, int gentype)

Typestr converter

::

int

PyArray\_DescrConverter(PyObject \*obj, PyArray\_Descr \*\*at)

Get typenum from an object -- None goes to NPY\_DEFAULT\_TYPE

This function takes a Python object representing a type and converts it

to a the correct PyArray\_Descr \* structure to describe the type.

Many objects can be used to represent a data-type which in NumPy is

quite a flexible concept.

This is the central code that converts Python objects to

Type-descriptor objects that are used throughout numpy.

Returns a new reference in \*at, but the returned should not be

modified as it may be one of the canonical immutable objects or

a reference to the input obj.

::

int

PyArray\_DescrConverter2(PyObject \*obj, PyArray\_Descr \*\*at)

Get typenum from an object -- None goes to NULL

::

int

PyArray\_IntpConverter(PyObject \*obj, PyArray\_Dims \*seq)

Get intp chunk from sequence

This function takes a Python sequence object and allocates and

fills in an intp array with the converted values.

Remember to free the pointer seq.ptr when done using

PyDimMem\_FREE(seq.ptr)\*\*

::

int

PyArray\_BufferConverter(PyObject \*obj, PyArray\_Chunk \*buf)

Get buffer chunk from object

this function takes a Python object which exposes the (single-segment)

buffer interface and returns a pointer to the data segment

You should increment the reference count by one of buf->base

if you will hang on to a reference

You only get a borrowed reference to the object. Do not free the

memory...

::

int

PyArray\_AxisConverter(PyObject \*obj, int \*axis)

Get axis from an object (possibly None) -- a converter function,

See also PyArray\_ConvertMultiAxis, which also handles a tuple of axes.

::

int

PyArray\_BoolConverter(PyObject \*object, npy\_bool \*val)

Convert an object to true / false

::

int

PyArray\_ByteorderConverter(PyObject \*obj, char \*endian)

Convert object to endian

::

int

PyArray\_OrderConverter(PyObject \*object, NPY\_ORDER \*val)

Convert an object to FORTRAN / C / ANY / KEEP

::

unsigned char

PyArray\_EquivTypes(PyArray\_Descr \*type1, PyArray\_Descr \*type2)

This function returns true if the two typecodes are

equivalent (same basic kind and same itemsize).

::

PyObject \*

PyArray\_Zeros(int nd, npy\_intp const \*dims, PyArray\_Descr \*type, int

is\_f\_order)

Zeros

steals a reference to type. On failure or when dtype->subarray is

true, dtype will be decrefed.

accepts NULL type

::

PyObject \*

PyArray\_Empty(int nd, npy\_intp const \*dims, PyArray\_Descr \*type, int

is\_f\_order)

Empty

accepts NULL type

steals a reference to type

::

PyObject \*

PyArray\_Where(PyObject \*condition, PyObject \*x, PyObject \*y)

Where

::

PyObject \*

PyArray\_Arange(double start, double stop, double step, int type\_num)

Arange,

::

PyObject \*

PyArray\_ArangeObj(PyObject \*start, PyObject \*stop, PyObject

\*step, PyArray\_Descr \*dtype)

ArangeObj,

this doesn't change the references

::

int

PyArray\_SortkindConverter(PyObject \*obj, NPY\_SORTKIND \*sortkind)

Convert object to sort kind

::

PyObject \*

PyArray\_LexSort(PyObject \*sort\_keys, int axis)

LexSort an array providing indices that will sort a collection of arrays

lexicographically. The first key is sorted on first, followed by the second key

-- requires that arg"merge"sort is available for each sort\_key

Returns an index array that shows the indexes for the lexicographic sort along

the given axis.

::

PyObject \*

PyArray\_Round(PyArrayObject \*a, int decimals, PyArrayObject \*out)

Round

::

unsigned char

PyArray\_EquivTypenums(int typenum1, int typenum2)

::

int

PyArray\_RegisterDataType(PyArray\_Descr \*descr)

Register Data type

Does not change the reference count of descr

::

int

PyArray\_RegisterCastFunc(PyArray\_Descr \*descr, int

totype, PyArray\_VectorUnaryFunc \*castfunc)

Register Casting Function

Replaces any function currently stored.

::

int

PyArray\_RegisterCanCast(PyArray\_Descr \*descr, int

totype, NPY\_SCALARKIND scalar)

Register a type number indicating that a descriptor can be cast

to it safely

::

void

PyArray\_InitArrFuncs(PyArray\_ArrFuncs \*f)

Initialize arrfuncs to NULL

::

PyObject \*

PyArray\_IntTupleFromIntp(int len, npy\_intp \*vals)

PyArray\_IntTupleFromIntp

::

int

PyArray\_TypeNumFromName(char \*str)

::

int

PyArray\_ClipmodeConverter(PyObject \*object, NPY\_CLIPMODE \*val)

Convert an object to NPY\_RAISE / NPY\_CLIP / NPY\_WRAP

::

int

PyArray\_OutputConverter(PyObject \*object, PyArrayObject \*\*address)

Useful to pass as converter function for O& processing in

PyArgs\_ParseTuple for output arrays

::

PyObject \*

PyArray\_BroadcastToShape(PyObject \*obj, npy\_intp \*dims, int nd)

Get Iterator broadcast to a particular shape

::

void

\_PyArray\_SigintHandler(int signum)

::

void\*

\_PyArray\_GetSigintBuf(void )

::

int

PyArray\_DescrAlignConverter(PyObject \*obj, PyArray\_Descr \*\*at)

Get type-descriptor from an object forcing alignment if possible

None goes to DEFAULT type.

any object with the .fields attribute and/or .itemsize attribute (if the

.fields attribute does not give the total size -- i.e. a partial record

naming). If itemsize is given it must be >= size computed from fields

The .fields attribute must return a convertible dictionary if present.

Result inherits from NPY\_VOID.

::

int

PyArray\_DescrAlignConverter2(PyObject \*obj, PyArray\_Descr \*\*at)

Get type-descriptor from an object forcing alignment if possible

None goes to NULL.

::

int

PyArray\_SearchsideConverter(PyObject \*obj, void \*addr)

Convert object to searchsorted side

::

PyObject \*

PyArray\_CheckAxis(PyArrayObject \*arr, int \*axis, int flags)

PyArray\_CheckAxis

check that axis is valid

convert 0-d arrays to 1-d arrays

::

npy\_intp

PyArray\_OverflowMultiplyList(npy\_intp \*l1, int n)

Multiply a List of Non-negative numbers with over-flow detection.

::

int

PyArray\_CompareString(const char \*s1, const char \*s2, size\_t len)

::

PyObject\*

PyArray\_MultiIterFromObjects(PyObject \*\*mps, int n, int nadd, ... )

Get MultiIterator from array of Python objects and any additional

PyObject \*\*mps - array of PyObjects

int n - number of PyObjects in the array

int nadd - number of additional arrays to include in the iterator.

Returns a multi-iterator object.

::

int

PyArray\_GetEndianness(void )

::

unsigned int

PyArray\_GetNDArrayCFeatureVersion(void )

Returns the built-in (at compilation time) C API version

::

PyObject \*

PyArray\_Correlate2(PyObject \*op1, PyObject \*op2, int mode)

correlate(a1,a2,mode)

This function computes the usual correlation (correlate(a1, a2) !=

correlate(a2, a1), and conjugate the second argument for complex inputs

::

PyObject\*

PyArray\_NeighborhoodIterNew(PyArrayIterObject \*x, const npy\_intp

\*bounds, int mode, PyArrayObject\*fill)

A Neighborhood Iterator object.

::

void

PyArray\_SetDatetimeParseFunction(PyObject \*NPY\_UNUSED(op) )

This function is scheduled to be removed

TO BE REMOVED - NOT USED INTERNALLY.

::

void

PyArray\_DatetimeToDatetimeStruct(npy\_datetime NPY\_UNUSED(val)

, NPY\_DATETIMEUNIT NPY\_UNUSED(fr)

, npy\_datetimestruct \*result)

Fill the datetime struct from the value and resolution unit.

TO BE REMOVED - NOT USED INTERNALLY.

::

void

PyArray\_TimedeltaToTimedeltaStruct(npy\_timedelta NPY\_UNUSED(val)

, NPY\_DATETIMEUNIT NPY\_UNUSED(fr)

, npy\_timedeltastruct \*result)

Fill the timedelta struct from the timedelta value and resolution unit.

TO BE REMOVED - NOT USED INTERNALLY.

::

npy\_datetime

PyArray\_DatetimeStructToDatetime(NPY\_DATETIMEUNIT NPY\_UNUSED(fr)

, npy\_datetimestruct \*NPY\_UNUSED(d) )

Create a datetime value from a filled datetime struct and resolution unit.

TO BE REMOVED - NOT USED INTERNALLY.

::

npy\_datetime

PyArray\_TimedeltaStructToTimedelta(NPY\_DATETIMEUNIT NPY\_UNUSED(fr)

, npy\_timedeltastruct

\*NPY\_UNUSED(d) )

Create a timdelta value from a filled timedelta struct and resolution unit.

TO BE REMOVED - NOT USED INTERNALLY.

::

NpyIter \*

NpyIter\_New(PyArrayObject \*op, npy\_uint32 flags, NPY\_ORDER

order, NPY\_CASTING casting, PyArray\_Descr\*dtype)

Allocate a new iterator for one array object.

::

NpyIter \*

NpyIter\_MultiNew(int nop, PyArrayObject \*\*op\_in, npy\_uint32

flags, NPY\_ORDER order, NPY\_CASTING

casting, npy\_uint32 \*op\_flags, PyArray\_Descr

\*\*op\_request\_dtypes)

Allocate a new iterator for more than one array object, using

standard NumPy broadcasting rules and the default buffer size.

::

NpyIter \*

NpyIter\_AdvancedNew(int nop, PyArrayObject \*\*op\_in, npy\_uint32

flags, NPY\_ORDER order, NPY\_CASTING

casting, npy\_uint32 \*op\_flags, PyArray\_Descr

\*\*op\_request\_dtypes, int oa\_ndim, int

\*\*op\_axes, npy\_intp \*itershape, npy\_intp

buffersize)

Allocate a new iterator for multiple array objects, and advanced

options for controlling the broadcasting, shape, and buffer size.

::

NpyIter \*

NpyIter\_Copy(NpyIter \*iter)

Makes a copy of the iterator

::

int

NpyIter\_Deallocate(NpyIter \*iter)

Deallocate an iterator

::

npy\_bool

NpyIter\_HasDelayedBufAlloc(NpyIter \*iter)

Whether the buffer allocation is being delayed

::

npy\_bool

NpyIter\_HasExternalLoop(NpyIter \*iter)

Whether the iterator handles the inner loop

::

int

NpyIter\_EnableExternalLoop(NpyIter \*iter)

Removes the inner loop handling (so HasExternalLoop returns true)

::

npy\_intp \*

NpyIter\_GetInnerStrideArray(NpyIter \*iter)

Get the array of strides for the inner loop (when HasExternalLoop is true)

This function may be safely called without holding the Python GIL.

::

npy\_intp \*

NpyIter\_GetInnerLoopSizePtr(NpyIter \*iter)

Get a pointer to the size of the inner loop (when HasExternalLoop is true)

This function may be safely called without holding the Python GIL.

::

int

NpyIter\_Reset(NpyIter \*iter, char \*\*errmsg)

Resets the iterator to its initial state

If errmsg is non-NULL, it should point to a variable which will

receive the error message, and no Python exception will be set.

This is so that the function can be called from code not holding

the GIL.

::

int

NpyIter\_ResetBasePointers(NpyIter \*iter, char \*\*baseptrs, char

\*\*errmsg)

Resets the iterator to its initial state, with new base data pointers.

This function requires great caution.

If errmsg is non-NULL, it should point to a variable which will

receive the error message, and no Python exception will be set.

This is so that the function can be called from code not holding

the GIL.

::

int

NpyIter\_ResetToIterIndexRange(NpyIter \*iter, npy\_intp istart, npy\_intp

iend, char \*\*errmsg)

Resets the iterator to a new iterator index range

If errmsg is non-NULL, it should point to a variable which will

receive the error message, and no Python exception will be set.

This is so that the function can be called from code not holding

the GIL.

::

int

NpyIter\_GetNDim(NpyIter \*iter)

Gets the number of dimensions being iterated

::

int

NpyIter\_GetNOp(NpyIter \*iter)

Gets the number of operands being iterated

::

NpyIter\_IterNextFunc \*

NpyIter\_GetIterNext(NpyIter \*iter, char \*\*errmsg)

Compute the specialized iteration function for an iterator

If errmsg is non-NULL, it should point to a variable which will

receive the error message, and no Python exception will be set.

This is so that the function can be called from code not holding

the GIL.

::

npy\_intp

NpyIter\_GetIterSize(NpyIter \*iter)

Gets the number of elements being iterated

::

void

NpyIter\_GetIterIndexRange(NpyIter \*iter, npy\_intp \*istart, npy\_intp

\*iend)

Gets the range of iteration indices being iterated

::

npy\_intp

NpyIter\_GetIterIndex(NpyIter \*iter)

Gets the current iteration index

::

int

NpyIter\_GotoIterIndex(NpyIter \*iter, npy\_intp iterindex)

Sets the iterator position to the specified iterindex,

which matches the iteration order of the iterator.

Returns NPY\_SUCCEED on success, NPY\_FAIL on failure.

::

npy\_bool

NpyIter\_HasMultiIndex(NpyIter \*iter)

Whether the iterator is tracking a multi-index

::

int

NpyIter\_GetShape(NpyIter \*iter, npy\_intp \*outshape)

Gets the broadcast shape if a multi-index is being tracked by the iterator,

otherwise gets the shape of the iteration as Fortran-order

(fastest-changing index first).

The reason Fortran-order is returned when a multi-index

is not enabled is that this is providing a direct view into how

the iterator traverses the n-dimensional space. The iterator organizes

its memory from fastest index to slowest index, and when

a multi-index is enabled, it uses a permutation to recover the original

order.

Returns NPY\_SUCCEED or NPY\_FAIL.

::

NpyIter\_GetMultiIndexFunc \*

NpyIter\_GetGetMultiIndex(NpyIter \*iter, char \*\*errmsg)

Compute a specialized get\_multi\_index function for the iterator

If errmsg is non-NULL, it should point to a variable which will

receive the error message, and no Python exception will be set.

This is so that the function can be called from code not holding

the GIL.

::

int

NpyIter\_GotoMultiIndex(NpyIter \*iter, npy\_intp const \*multi\_index)

Sets the iterator to the specified multi-index, which must have the

correct number of entries for 'ndim'. It is only valid

when NPY\_ITER\_MULTI\_INDEX was passed to the constructor. This operation

fails if the multi-index is out of bounds.

Returns NPY\_SUCCEED on success, NPY\_FAIL on failure.

::

int

NpyIter\_RemoveMultiIndex(NpyIter \*iter)

Removes multi-index support from an iterator.

Returns NPY\_SUCCEED or NPY\_FAIL.

::

npy\_bool

NpyIter\_HasIndex(NpyIter \*iter)

Whether the iterator is tracking an index

::

npy\_bool

NpyIter\_IsBuffered(NpyIter \*iter)

Whether the iterator is buffered

::

npy\_bool

NpyIter\_IsGrowInner(NpyIter \*iter)

Whether the inner loop can grow if buffering is unneeded

::

npy\_intp

NpyIter\_GetBufferSize(NpyIter \*iter)

Gets the size of the buffer, or 0 if buffering is not enabled

::

npy\_intp \*

NpyIter\_GetIndexPtr(NpyIter \*iter)

Get a pointer to the index, if it is being tracked

::

int

NpyIter\_GotoIndex(NpyIter \*iter, npy\_intp flat\_index)

If the iterator is tracking an index, sets the iterator

to the specified index.

Returns NPY\_SUCCEED on success, NPY\_FAIL on failure.

::

char \*\*

NpyIter\_GetDataPtrArray(NpyIter \*iter)

Get the array of data pointers (1 per object being iterated)

This function may be safely called without holding the Python GIL.

::

PyArray\_Descr \*\*

NpyIter\_GetDescrArray(NpyIter \*iter)

Get the array of data type pointers (1 per object being iterated)

::

PyArrayObject \*\*

NpyIter\_GetOperandArray(NpyIter \*iter)

Get the array of objects being iterated

::

PyArrayObject \*

NpyIter\_GetIterView(NpyIter \*iter, npy\_intp i)

Returns a view to the i-th object with the iterator's internal axes

::

void

NpyIter\_GetReadFlags(NpyIter \*iter, char \*outreadflags)

Gets an array of read flags (1 per object being iterated)

::

void

NpyIter\_GetWriteFlags(NpyIter \*iter, char \*outwriteflags)

Gets an array of write flags (1 per object being iterated)

::

void

NpyIter\_DebugPrint(NpyIter \*iter)

For debugging

::

npy\_bool

NpyIter\_IterationNeedsAPI(NpyIter \*iter)

Whether the iteration loop, and in particular the iternext()

function, needs API access. If this is true, the GIL must

be retained while iterating.

::

void

NpyIter\_GetInnerFixedStrideArray(NpyIter \*iter, npy\_intp \*out\_strides)

Get an array of strides which are fixed. Any strides which may

change during iteration receive the value NPY\_MAX\_INTP. Once

the iterator is ready to iterate, call this to get the strides

which will always be fixed in the inner loop, then choose optimized

inner loop functions which take advantage of those fixed strides.

This function may be safely called without holding the Python GIL.

::

int

NpyIter\_RemoveAxis(NpyIter \*iter, int axis)

Removes an axis from iteration. This requires that NPY\_ITER\_MULTI\_INDEX

was set for iterator creation, and does not work if buffering is

enabled. This function also resets the iterator to its initial state.

Returns NPY\_SUCCEED or NPY\_FAIL.

::

npy\_intp \*

NpyIter\_GetAxisStrideArray(NpyIter \*iter, int axis)

Gets the array of strides for the specified axis.

If the iterator is tracking a multi-index, gets the strides

for the axis specified, otherwise gets the strides for

the iteration axis as Fortran order (fastest-changing axis first).

Returns NULL if an error occurs.

::

npy\_bool

NpyIter\_RequiresBuffering(NpyIter \*iter)

Whether the iteration could be done with no buffering.

::

char \*\*

NpyIter\_GetInitialDataPtrArray(NpyIter \*iter)

Get the array of data pointers (1 per object being iterated),

directly into the arrays (never pointing to a buffer), for starting

unbuffered iteration. This always returns the addresses for the

iterator position as reset to iterator index 0.

These pointers are different from the pointers accepted by

NpyIter\_ResetBasePointers, because the direction along some

axes may have been reversed, requiring base offsets.

This function may be safely called without holding the Python GIL.

::

int

NpyIter\_CreateCompatibleStrides(NpyIter \*iter, npy\_intp

itemsize, npy\_intp \*outstrides)

Builds a set of strides which are the same as the strides of an

output array created using the NPY\_ITER\_ALLOCATE flag, where NULL

was passed for op\_axes. This is for data packed contiguously,

but not necessarily in C or Fortran order. This should be used

together with NpyIter\_GetShape and NpyIter\_GetNDim.

A use case for this function is to match the shape and layout of

the iterator and tack on one or more dimensions. For example,

in order to generate a vector per input value for a numerical gradient,

you pass in ndim\*itemsize for itemsize, then add another dimension to

the end with size ndim and stride itemsize. To do the Hessian matrix,

you do the same thing but add two dimensions, or take advantage of

the symmetry and pack it into 1 dimension with a particular encoding.

This function may only be called if the iterator is tracking a multi-index

and if NPY\_ITER\_DONT\_NEGATE\_STRIDES was used to prevent an axis from

being iterated in reverse order.

If an array is created with this method, simply adding 'itemsize'

for each iteration will traverse the new array matching the

iterator.

Returns NPY\_SUCCEED or NPY\_FAIL.

::

int

PyArray\_CastingConverter(PyObject \*obj, NPY\_CASTING \*casting)

Convert any Python object, \*obj\*, to an NPY\_CASTING enum.

::

npy\_intp

PyArray\_CountNonzero(PyArrayObject \*self)

Counts the number of non-zero elements in the array.

Returns -1 on error.

::

PyArray\_Descr \*

PyArray\_PromoteTypes(PyArray\_Descr \*type1, PyArray\_Descr \*type2)

Produces the smallest size and lowest kind type to which both

input types can be cast.

::

PyArray\_Descr \*

PyArray\_MinScalarType(PyArrayObject \*arr)

If arr is a scalar (has 0 dimensions) with a built-in number data type,

finds the smallest type size/kind which can still represent its data.

Otherwise, returns the array's data type.

::

PyArray\_Descr \*

PyArray\_ResultType(npy\_intp narrs, PyArrayObject \*\*arr, npy\_intp

ndtypes, PyArray\_Descr \*\*dtypes)

Produces the result type of a bunch of inputs, using the UFunc

type promotion rules. Use this function when you have a set of

input arrays, and need to determine an output array dtype.

If all the inputs are scalars (have 0 dimensions) or the maximum "kind"

of the scalars is greater than the maximum "kind" of the arrays, does

a regular type promotion.

Otherwise, does a type promotion on the MinScalarType

of all the inputs. Data types passed directly are treated as array

types.

::

npy\_bool

PyArray\_CanCastArrayTo(PyArrayObject \*arr, PyArray\_Descr

\*to, NPY\_CASTING casting)

Returns 1 if the array object may be cast to the given data type using

the casting rule, 0 otherwise. This differs from PyArray\_CanCastTo in

that it handles scalar arrays (0 dimensions) specially, by checking

their value.

::

npy\_bool

PyArray\_CanCastTypeTo(PyArray\_Descr \*from, PyArray\_Descr

\*to, NPY\_CASTING casting)

Returns true if data of type 'from' may be cast to data of type

'to' according to the rule 'casting'.

::

PyArrayObject \*

PyArray\_EinsteinSum(char \*subscripts, npy\_intp nop, PyArrayObject

\*\*op\_in, PyArray\_Descr \*dtype, NPY\_ORDER

order, NPY\_CASTING casting, PyArrayObject \*out)

This function provides summation of array elements according to

the Einstein summation convention. For example:

- trace(a) -> einsum("ii", a)

- transpose(a) -> einsum("ji", a)

- multiply(a,b) -> einsum(",", a, b)

- inner(a,b) -> einsum("i,i", a, b)

- outer(a,b) -> einsum("i,j", a, b)

- matvec(a,b) -> einsum("ij,j", a, b)

- matmat(a,b) -> einsum("ij,jk", a, b)

subscripts: The string of subscripts for einstein summation.

nop: The number of operands

op\_in: The array of operands

dtype: Either NULL, or the data type to force the calculation as.

order: The order for the calculation/the output axes.

casting: What kind of casts should be permitted.

out: Either NULL, or an array into which the output should be placed.

By default, the labels get placed in alphabetical order

at the end of the output. So, if c = einsum("i,j", a, b)

then c[i,j] == a[i]\*b[j], but if c = einsum("j,i", a, b)

then c[i,j] = a[j]\*b[i].

Alternatively, you can control the output order or prevent

an axis from being summed/force an axis to be summed by providing

indices for the output. This allows us to turn 'trace' into

'diag', for example.

- diag(a) -> einsum("ii->i", a)

- sum(a, axis=0) -> einsum("i...->", a)

Subscripts at the beginning and end may be specified by

putting an ellipsis "..." in the middle. For example,

the function einsum("i...i", a) takes the diagonal of

the first and last dimensions of the operand, and

einsum("ij...,jk...->ik...") takes the matrix product using

the first two indices of each operand instead of the last two.

When there is only one operand, no axes being summed, and

no output parameter, this function returns a view

into the operand instead of making a copy.

::

PyObject \*

PyArray\_NewLikeArray(PyArrayObject \*prototype, NPY\_ORDER

order, PyArray\_Descr \*dtype, int subok)

Creates a new array with the same shape as the provided one,

with possible memory layout order and data type changes.

prototype - The array the new one should be like.

order - NPY\_CORDER - C-contiguous result.

NPY\_FORTRANORDER - Fortran-contiguous result.

NPY\_ANYORDER - Fortran if prototype is Fortran, C otherwise.

NPY\_KEEPORDER - Keeps the axis ordering of prototype.

dtype - If not NULL, overrides the data type of the result.

subok - If 1, use the prototype's array subtype, otherwise

always create a base-class array.

NOTE: If dtype is not NULL, steals the dtype reference. On failure or when

dtype->subarray is true, dtype will be decrefed.

::

int

PyArray\_GetArrayParamsFromObject(PyObject \*op, PyArray\_Descr

\*requested\_dtype, npy\_bool

writeable, PyArray\_Descr

\*\*out\_dtype, int \*out\_ndim, npy\_intp

\*out\_dims, PyArrayObject

\*\*out\_arr, PyObject \*context)

Retrieves the array parameters for viewing/converting an arbitrary

PyObject\* to a NumPy array. This allows the "innate type and shape"

of Python list-of-lists to be discovered without

actually converting to an array.

In some cases, such as structured arrays and the \_\_array\_\_ interface,

a data type needs to be used to make sense of the object. When

this is needed, provide a Descr for 'requested\_dtype', otherwise

provide NULL. This reference is not stolen. Also, if the requested

dtype doesn't modify the interpretation of the input, out\_dtype will

still get the "innate" dtype of the object, not the dtype passed

in 'requested\_dtype'.

If writing to the value in 'op' is desired, set the boolean

'writeable' to 1. This raises an error when 'op' is a scalar, list

of lists, or other non-writeable 'op'.

Result: When success (0 return value) is returned, either out\_arr

is filled with a non-NULL PyArrayObject and

the rest of the parameters are untouched, or out\_arr is

filled with NULL, and the rest of the parameters are

filled.

Typical usage:

PyArrayObject \*arr = NULL;

PyArray\_Descr \*dtype = NULL;

int ndim = 0;

npy\_intp dims[NPY\_MAXDIMS];

if (PyArray\_GetArrayParamsFromObject(op, NULL, 1, &dtype,

&ndim, dims, &arr, NULL) < 0) {

return NULL;

}

if (arr == NULL) {

... validate/change dtype, validate flags, ndim, etc ...

// Could make custom strides here too

arr = PyArray\_NewFromDescr(&PyArray\_Type, dtype, ndim,

dims, NULL,

is\_f\_order ? NPY\_ARRAY\_F\_CONTIGUOUS : 0,

NULL);

if (arr == NULL) {

return NULL;

}

if (PyArray\_CopyObject(arr, op) < 0) {

Py\_DECREF(arr);

return NULL;

}

}

else {

... in this case the other parameters weren't filled, just

validate and possibly copy arr itself ...

}

... use arr ...

::

int

PyArray\_ConvertClipmodeSequence(PyObject \*object, NPY\_CLIPMODE

\*modes, int n)

Convert an object to an array of n NPY\_CLIPMODE values.

This is intended to be used in functions where a different mode

could be applied to each axis, like in ravel\_multi\_index.

::

PyObject \*

PyArray\_MatrixProduct2(PyObject \*op1, PyObject

\*op2, PyArrayObject\*out)

Numeric.matrixproduct2(a,v,out)

just like inner product but does the swapaxes stuff on the fly

::

npy\_bool

NpyIter\_IsFirstVisit(NpyIter \*iter, int iop)

Checks to see whether this is the first time the elements

of the specified reduction operand which the iterator points at are

being seen for the first time. The function returns

a reasonable answer for reduction operands and when buffering is

disabled. The answer may be incorrect for buffered non-reduction

operands.

This function is intended to be used in EXTERNAL\_LOOP mode only,

and will produce some wrong answers when that mode is not enabled.

If this function returns true, the caller should also

check the inner loop stride of the operand, because if

that stride is 0, then only the first element of the innermost

external loop is being visited for the first time.

WARNING: For performance reasons, 'iop' is not bounds-checked,

it is not confirmed that 'iop' is actually a reduction

operand, and it is not confirmed that EXTERNAL\_LOOP

mode is enabled. These checks are the responsibility of

the caller, and should be done outside of any inner loops.

::

int

PyArray\_SetBaseObject(PyArrayObject \*arr, PyObject \*obj)

Sets the 'base' attribute of the array. This steals a reference

to 'obj'.

Returns 0 on success, -1 on failure.

::

void

PyArray\_CreateSortedStridePerm(int ndim, npy\_intp const

\*strides, npy\_stride\_sort\_item

\*out\_strideperm)

This function populates the first ndim elements

of strideperm with sorted descending by their absolute values.

For example, the stride array (4, -2, 12) becomes

[(2, 12), (0, 4), (1, -2)].

::

void

PyArray\_RemoveAxesInPlace(PyArrayObject \*arr, const npy\_bool \*flags)

Removes the axes flagged as True from the array,

modifying it in place. If an axis flagged for removal

has a shape entry bigger than one, this effectively selects

index zero for that axis.

WARNING: If an axis flagged for removal has a shape equal to zero,

the array will point to invalid memory. The caller must

validate this!

If an axis flagged for removal has a shape larger than one,

the aligned flag (and in the future the contiguous flags),

may need explicit update.

(check also NPY\_RELAXED\_STRIDES\_CHECKING)

For example, this can be used to remove the reduction axes

from a reduction result once its computation is complete.

::

void

PyArray\_DebugPrint(PyArrayObject \*obj)

Prints the raw data of the ndarray in a form useful for debugging

low-level C issues.

::

int

PyArray\_FailUnlessWriteable(PyArrayObject \*obj, const char \*name)

This function does nothing if obj is writeable, and raises an exception

(and returns -1) if obj is not writeable. It may also do other

house-keeping, such as issuing warnings on arrays which are transitioning

to become views. Always call this function at some point before writing to

an array.

'name' is a name for the array, used to give better error

messages. Something like "assignment destination", "output array", or even

just "array".

::

int

PyArray\_SetUpdateIfCopyBase(PyArrayObject \*arr, PyArrayObject \*base)

Precondition: 'arr' is a copy of 'base' (though possibly with different

strides, ordering, etc.). This function sets the UPDATEIFCOPY flag and the

->base pointer on 'arr', so that when 'arr' is destructed, it will copy any

changes back to 'base'. DEPRECATED, use PyArray\_SetWritebackIfCopyBase

Steals a reference to 'base'.

Returns 0 on success, -1 on failure.

::

void \*

PyDataMem\_NEW(size\_t size)

Allocates memory for array data.

::

void

PyDataMem\_FREE(void \*ptr)

Free memory for array data.

::

void \*

PyDataMem\_RENEW(void \*ptr, size\_t size)

Reallocate/resize memory for array data.

::

PyDataMem\_EventHookFunc \*

PyDataMem\_SetEventHook(PyDataMem\_EventHookFunc \*newhook, void

\*user\_data, void \*\*old\_data)

Sets the allocation event hook for numpy array data.

Takes a PyDataMem\_EventHookFunc \*, which has the signature:

void hook(void \*old, void \*new, size\_t size, void \*user\_data).

Also takes a void \*user\_data, and void \*\*old\_data.

Returns a pointer to the previous hook or NULL. If old\_data is

non-NULL, the previous user\_data pointer will be copied to it.

If not NULL, hook will be called at the end of each PyDataMem\_NEW/FREE/RENEW:

result = PyDataMem\_NEW(size) -> (\*hook)(NULL, result, size, user\_data)

PyDataMem\_FREE(ptr) -> (\*hook)(ptr, NULL, 0, user\_data)

result = PyDataMem\_RENEW(ptr, size) -> (\*hook)(ptr, result, size, user\_data)

When the hook is called, the GIL will be held by the calling

thread. The hook should be written to be reentrant, if it performs

operations that might cause new allocation events (such as the

creation/destruction numpy objects, or creating/destroying Python

objects which might cause a gc)

::

void

PyArray\_MapIterSwapAxes(PyArrayMapIterObject \*mit, PyArrayObject

\*\*ret, int getmap)

::

PyObject \*

PyArray\_MapIterArray(PyArrayObject \*a, PyObject \*index)

Use advanced indexing to iterate an array.

::

void

PyArray\_MapIterNext(PyArrayMapIterObject \*mit)

This function needs to update the state of the map iterator

and point mit->dataptr to the memory-location of the next object

Note that this function never handles an extra operand but provides

compatibility for an old (exposed) API.

::

int

PyArray\_Partition(PyArrayObject \*op, PyArrayObject \*ktharray, int

axis, NPY\_SELECTKIND which)

Partition an array in-place

::

PyObject \*

PyArray\_ArgPartition(PyArrayObject \*op, PyArrayObject \*ktharray, int

axis, NPY\_SELECTKIND which)

ArgPartition an array

::

int

PyArray\_SelectkindConverter(PyObject \*obj, NPY\_SELECTKIND \*selectkind)

Convert object to select kind

::

void \*

PyDataMem\_NEW\_ZEROED(size\_t size, size\_t elsize)

Allocates zeroed memory for array data.

::

int

PyArray\_CheckAnyScalarExact(PyObject \*obj)

return true an object is exactly a numpy scalar

::

PyObject \*

PyArray\_MapIterArrayCopyIfOverlap(PyArrayObject \*a, PyObject

\*index, int

copy\_if\_overlap, PyArrayObject

\*extra\_op)

Same as PyArray\_MapIterArray, but:

If copy\_if\_overlap != 0, check if `a` has memory overlap with any of the

arrays in `index` and with `extra\_op`. If yes, make copies as appropriate

to avoid problems if `a` is modified during the iteration.

`iter->array` may contain a copied array (UPDATEIFCOPY/WRITEBACKIFCOPY set).

::

int

PyArray\_ResolveWritebackIfCopy(PyArrayObject \*self)

If WRITEBACKIFCOPY and self has data, reset the base WRITEABLE flag,

copy the local data to base, release the local data, and set flags

appropriately. Return 0 if not relevant, 1 if success, < 0 on failure

::

int

PyArray\_SetWritebackIfCopyBase(PyArrayObject \*arr, PyArrayObject

\*base)

Precondition: 'arr' is a copy of 'base' (though possibly with different

strides, ordering, etc.). This function sets the WRITEBACKIFCOPY flag and the

->base pointer on 'arr', call PyArray\_ResolveWritebackIfCopy to copy any

changes back to 'base' before deallocating the array.

Steals a reference to 'base'.

Returns 0 on success, -1 on failure.