CPMpy

Release 0.5

Tias Guns

Feb 26, 2021

CONTENTS

Welcome to CpMPy. Licensed under the MIT License.

CpMPy is a numpy-based light-weight Python library for conveniently modeling constraint problems in Python. It aims to connect to common constraint solving systems that have a Python API, such as MiniZinc (with solvers gecode, chuffed, ortools, picatsat, etc), or-tools through its Python API and more.

It is inspired by CVXpy, SciPy and Numberjack, and as most modern scientific Python tools, it uses numpy arrays as basic data structure.

A longer description of its motivation and architecture is in `pdf`.

The software is in ALPHA state, and more of a proof-of-concept really. Do send suggestions, additions, API changes, or even reuse some of these ideas in your own project!

CHAPTER

ONE

INSTALL THE LIBRARY

CHAPTER

TWO

DOCUMENTATION

2.1 Constraint Programming: a quick CPMpy prototype

2.1.1 Constraint Programming

Many real-life decisions involve a large number of options. To decide if a problem is feasible or finding the best one amongst all the options is hard task to do by hand. In other words, to enumerate all the possible combinations of single decisions and evaluate them is infeasible in practice. To avoid this "*brute force*" approach, the paradigm of **constraint programming (CP)** allow us to:

1. Model relationships between single decisions smartly
2. Give an answer efficiently.

A **constraint satisfaction problem (CSP)** consists of a set of variables and constraints stablishing relationships between them. Each variable has a finite of possible values (its domain). The goal is to assign values to the variables in its domains satisfying all the constraints. A more general version, called **constraint optimization programming (C0P)**, finds amongst all the feasible solutions the one that optimizes some measure, called 'objective function'.

What is necessary to model a CP?

A typical CP is defined by the following elements:

Variables: define variables and domain. Types of domains for different types of variables.

Constraints: Short summary of constraints

Moreover, if we want to model an optimization problem we also need an objective function.

Example

A cryptarithmetic puzzle is a mathematical exercise where the digits of some numbers are represented by letters (or symbols). Each letter represents a unique digit. The goal is to find the digits such that a given mathematical equation is verified.

For example, we aim to allocate to the letters S,E,N,D,M,O,R,Y a digit between 0 and 9, being all the letters allocated to a different digit and such that the expression:

SEND + MORE = MONEY

is satisfied. This problem lies into the setting of **constraint satisfaction problem (CSP)**. Here the variables are each letter S,E,N,D,M,O,R,Y and their domain is $\{0,1,2,\ldots,9\}$. The constraints represents the fact that

The cpmpy implementation for this CSP looks like:

```
from cpmpy import *
import numpy as np

# Construct the model
s,e,n,d,m,o,r,y = IntVar(0,9, 8)

constraint = []
constraint += [ alldifferent([s,e,n,d,m,o,r,y]) ]
constraint += [    sum(   [s,e,n,d] * np.flip(10**np.arange(4)) )
                 + sum(   [m,o,r,e] * np.flip(10**np.arange(4)) )
                == sum( [m,o,n,e,y] * np.flip(10**np.arange(5)) ) ]

model = Model(constraint)
print(model)

stats = model.solve()
print("  S,E,N,D =  ", [x.value() for x in [s,e,n,d]])
print("  M,O,R,E =  ", [x.value() for x in [m,o,r,e]])
print("M,O,N,E,Y =", [x.value() for x in [m,o,n,e,y]])
```

A possible feasible allocation/solution is

```
S,E,N,D =   [2, 8, 1, 7]
M,O,R,E =   [0, 3, 6, 8]
M,O,N,E,Y = [0, 3, 1, 8, 5]
```

Note that we can find an slightly different version of this problem by optimizing an objective function, for example, optimizing the number formed by the word MONEY:

To implement this COP, we need only to modify the Model statement by adding an objective function:

```
coefs  = np.flip(10**np.arange(5))
objective = np.dot([m,o,n,e,y],coefs)
model = Model(constraint, maximize = objective)
```

And the result will be:

```
S,E,N,D =   [9, 5, 6, 7]
M,O,R,E =   [1, 0, 8, 5]
M,O,N,E,Y = [1, 0, 6, 5, 2]
```

In the next, we are going to look in detail this example. But first you may want to look some references for a global overview of Constraint Programming.

References

To learn more about theory and practice of constraint programming you may want to check some references:

1. Rossi, F., Van Beek, P., & Walsh, T. (Eds.). (2006). Handbook of constraint programming. Elsevier.
2. Apt, K. (2003). Principles of constraint programming. Cambridge university press.

2.2 CPMpy's pipeline

CPMpy has two key parts:

1. the 'language' that allows expressing constraint programming problems,
2. a mechanism to translate this language to the API of solvers.

2.2.1 The language

When you write a CPMpy model (constraints and an objective), you use Python's operators (*,+,*sum*,-,~,|,& etc) on CPMpy variables, as well as CPMpy functions and global constraints.

CPMpy uses Python's operator overloading to build an expressions trees (see expression.py). From CPMpy's point of view, a constraint programming problem is a list of expression trees (each one representing a constraint) and an expression tree for the objective function. You can write very complex nested expressions (e.g. *(a | b) == ((x + y > 5) -> (c & d))*), the language itself has few restrictions.

CPMpy only does minor modifications to the expressions when building the expression trees, e.g. it removes constants when chaining operators (e.g. $x + 0 :: x$)

So the language offers acces to the high level expressions written by the user.

But solvers can't use this...

2.2.2 The mechanism

We have a number of staged transformations that the expression trees go through. These roughly correspond to different 'normal forms' as one would do in SAT, however, there are no 'normal forms' for constraint specifications as far as we are aware.

So far, we have the following 3 stages:

```
CPMpy expression trees -> flatten -> solver-specific transf -> solver API
|--------------------|    |-----|    |----------------------------------|
```

As said, CPMpy expression trees allow arbitrary nesting, but only modeling languages (like MiniZinc and XCSP3) allow that. So if we want to use a solver API directly, we need to 'flatten' the nested expressions first.

Then, every solver has its own API, as well as some peculiarities (e.g. or-tools only supports implication/half-reifiction ->, not standard double reification == (sometimes written as <->). These transformations are bundled into the solver-specific file in CPMpy.

2.2.3 Flattened 'normal form'

So that leaves the question, what is and is not allowed in this 'flattened' inbetween output?

Ideally, we can come up with a grammar that determines a formal normal form. By lack of that, here is a more informal description that is grammar-like.

```
Var = IntVar | BoolVar | Num
BoolVar = BoolVarImpl | NegBoolView | True | False
```

A variable is either an Integer decision variable, or a Boolean decision variable, or a numeric constant. For Boolean decision variables we have a special case: it is either an actual Boolean decision variable, or the negation of a Boolean decision variable (a negated 'view' on the variable), or the trivial True/False.

```
BaseCons = ("name", [Var])
```

A 'base' constraint is simply a name, with a list of variables (no nested expressions). This includes global constraints, but also conjunction, disjunction, equality, etc.

To support linear constraints and reification (equating the truth-value of a constraint to a Boolean variable), we allow a few cases where a comparison operator can have a base constraint as its left-hand side.

2.2.4 Special case 1, linear constraints:

We first define a linear expression as follows (weighted linear sum):

```
LinExpr = ("sum", ([Constant], [Var]))
```

This can be used in a linear constraint, or in the objective function:

```
Obj = Var | LinExpr
```

TODO: what about Max(), Min(), e.g. for makespan? Should be a standard operator?

A constraint that adds a comparison operator on a linear expression has two forms:

```
LinConsRel = ("==", (LinExpr, Var)) | ("!=", (LinExpr, Var))
```

So (dis)equality can have a variable (or a constant) as its right-hand side. Inequality comparison operators will not:

```
Op = ">" | ">=" | "<" | "<="
LinConsIne = (Op, (LinExpr, Num))
```

2.2.5 Special case 2, reification:

We first define the Boolean expressions that allow reification:

```
BoolExpr = ("and", [Var]) | ("or", [Var]) | LinConsRel | LinConsIne
```

TODO: some globals may also support reification... check and update here

Now, like linear constraints, in case of reification and implication, the left-hand side can be a simple Boolean expression with the right-hand side a Boolean variable:

```
Reif = ("==", (BoolExpr, BoolVar))
Impl = ("->", (BoolExpr, BoolVar))
```

2.3 Expressions

2.3.1 List of classes

Expression	each Expression is a function with a self.name and self.args (arguments) each Expression is considered to be a function whose value can be used in other expressions each Expression may implement: - boolexpr(): the Boolean form of the expression default: (expr == 1) override for Boolean expressions (preferably through __eq__, see Comparison) - value(): the value of the expression, default None
Operator	All kinds of operators on expressions, including mathematical and logical # convention for 2-ary operators: if one of the two is a constant, # it is stored first (as expr[0]), this eases weighted sum detection
Element	constraint Arr[X] = Y 'Y' here is optional, can use as function: Arr[X] + 3 <= Y
GlobalConstraint	

2.3.2 Module description

2.3.3 Module details

2.4 Model

2.4.1 List of classes

Model	CPMpy Model object, contains the constraint and objective expression trees

2.4.2 Module description

2.4.3 Module details

class cpmpy.model.**Model**(**args*, *minimize=None*, *maximize=None*)

CPMpy Model object, contains the constraint and objective expression trees

Arguments of constructor: *args: Expression object(s) or list(s) of Expression objects minimize: Expression object representing the objective to minimize maximize: Expression object representing the objective to maximize

At most one of minimize/maximize can be set, if none are set, it is assumed to be a satisfaction problem

make_and_from_list(*args*)

recursively reads a list of Expression and returns the 'And' conjunctive of the elements in the list

solve(*solver=None*)

Send the model to a solver and get the result

'solver': None (default) or in [s.name in get_supported_solvers()] or a SolverInterface object verifies that the solver is supported on the current system

Returns the computed output: - True if it is a satisfaction problem and it is satisfiable - False if it is a satisfaction problem and not satisfiable - [int] if it is an optimisation problem

status()

Returns the status of the latest solver run on this model

Status information includes exit status (optimality) and runtime.

Returns an object of `SolverStatus`

2.5 Variables

2.5.1 List of classes

`Operator`	All kinds of operators on expressions, including mathematical and logical # convention for 2-ary operators: if one of the two is a constant, # it is stored first (as expr[0]), this eases weighted sum detection
`Element`	constraint Arr[X] = Y 'Y' here is optional, can use as function: Arr[X] + 3 <= Y

2.5.2 Module description

2.5.3 Module details

class cpmpy.variables.**BoolVarImpl**(*lb=0*, *ub=1*)

class cpmpy.variables.**IntVarImpl**(*lb*, *ub*, *setname=True*)

class cpmpy.variables.**NDVarArray**(*shape*, ***kwargs*)

T

The transposed array.

Same as `self.transpose()`.

```
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

transpose

all(*axis=None*, *out=None*, *keepdims=False*, *, *where=True*)

Returns True if all elements evaluate to True.

Refer to *numpy.all* for full documentation.

numpy.all : equivalent function

any(*axis=None*, *out=None*, *keepdims=False*, *, *where=True*)
Returns True if any of the elements of *a* evaluate to True.

Refer to *numpy.any* for full documentation.

numpy.any : equivalent function

argmax(*axis=None*, *out=None*)
Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

numpy.argmax : equivalent function

argmin(*axis=None*, *out=None*)
Return indices of the minimum values along the given axis.

Refer to *numpy.argmin* for detailed documentation.

numpy.argmin : equivalent function

argpartition(*kth*, *axis=- 1*, *kind='introselect'*, *order=None*)
Returns the indices that would partition this array.

Refer to *numpy.argpartition* for full documentation.

New in version 1.8.0.

numpy.argpartition : equivalent function

argsort(*axis=- 1*, *kind=None*, *order=None*)
Returns the indices that would sort this array.

Refer to *numpy.argsort* for full documentation.

numpy.argsort : equivalent function

astype(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)
Copy of the array, cast to a specified type.

dtype [str or dtype] Typecode or data-type to which the array is cast.

order [{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

casting [{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

subok [bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

copy [bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the *dtype*, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

arr_t [ndarray] Unless *copy* is False and the other conditions for returning the input array are satisfied (see description for *copy* input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by *dtype*, *order*.

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

ComplexWarning When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

base

Base object if memory is from some other object.

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

byteswap(*inplace=False*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

inplace [bool, optional] If `True`, swap bytes in-place, default is `False`.

out [ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([  256,     1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

A.newbyteorder().byteswap() produces an array with the same values but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

choose(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to *numpy.choose* for full documentation.

numpy.choose : equivalent function

clip(*min=None*, *max=None*, *out=None*, ***kwargs*)

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to *numpy.clip* for full documentation.

numpy.clip : equivalent function

compress(*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to *numpy.compress* for full documentation.

numpy.compress : equivalent function

conj()

Complex-conjugate all elements.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

conjugate()

Return the complex conjugate, element-wise.

Refer to *numpy.conjugate* for full documentation.

numpy.conjugate : equivalent function

copy(*order='C'*)

Return a copy of the array.

order [{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar, but have different default values for their order= arguments.)

numpy.copy numpy.copyto

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

ctypes

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

None

c [Python object] Possessing attributes data, shape, strides, etc.

numpy.ctypeslib

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

_ctypes.**data**

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

_ctypes.**shape**

(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be *ctypes.c_int*, *ctypes.c_long*, or *ctypes.c_longlong* depending on the platform. The c_intp type is defined accordingly in *numpy.ctypeslib*. The ctypes array contains the shape of the underlying array.

_ctypes.**strides**

(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

_ctypes.**data_as**(*obj*)

Return the data pointer cast to a particular c-types object. For example, calling `self.`

_as_parameter_ is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

`_ctypes.`**shape_as**(*obj*)
Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

`_ctypes.`**strides_as**(*obj*)
Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

cumprod(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative product of the elements along the given axis.

Refer to *numpy.cumprod* for full documentation.

numpy.cumprod : equivalent function

cumsum(*axis=None*, *dtype=None*, *out=None*)
Return the cumulative sum of the elements along the given axis.

Refer to *numpy.cumsum* for full documentation.

numpy.cumsum : equivalent function

data
Python buffer object pointing to the start of the array's data.

diagonal(*offset=0*, *axis1=0*, *axis2=1*)
Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal()` for full documentation.

numpy.diagonal : equivalent function

dot(*b*, *out=None*)
Dot product of two arrays.

Refer to *numpy.dot* for full documentation.

numpy.dot : equivalent function

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2.,  2.],
       [2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8.,  8.],
       [8.,  8.]])
```

dtype

Data-type of the array's elements.

None

d : numpy dtype object

numpy.dtype

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

dump(*file*)

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

file [str or Path] A string naming the dump file.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

dumps()

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

None

fill(*value*)

Fill the array with a scalar value.

value [scalar] All elements of *a* will be assigned this value.

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

flags

Information about the memory layout of the array.

C_CONTIGUOUS (C) The data is in a single, C-style contiguous segment.

F_CONTIGUOUS (F) The data is in a single, Fortran-style contiguous segment.

OWNDATA (O) The array owns the memory it uses or borrows it from another object.

WRITEABLE (W) The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

ALIGNED (A) The data and all elements are aligned appropriately for the hardware.

WRITEBACKIFCOPY (X) This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

UPDATEIFCOPY (U) (Deprecated, use WRITEBACKIFCOPY) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

FNC F_CONTIGUOUS and not C_CONTIGUOUS.

FORC F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

BEHAVED (B) ALIGNED and WRITEABLE.

CARRAY (CA) BEHAVED and C_CONTIGUOUS.

FARRAY (FA) BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- WRITEBACKIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

flat

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

flatten : Return a copy of the array collapsed into one dimension.

flatiter

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

flatten(*order='C'*)

Return a copy of the array collapsed into one dimension.

order [{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

y [ndarray] A copy of the input array, flattened to one dimension.

ravel : Return a flattened array. flat : A 1-D flat iterator over the array.

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

getfield(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

dtype [str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

offset [int] Number of bytes to skip before beginning the element view.

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

imag

The imaginary part of the array.

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

item(**args*)

Copy an element of an array to a standard Python scalar and return it.

*args : Arguments (variable number and type)

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

z [Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

item is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
```

(continues on next page)

(continued from previous page)

```
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

itemset (**args*)

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

***args** [Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

Compared to indexing syntax, *itemset* provides some speed increase for placing a scalar into a particular location in an *ndarray*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *itemset* (and *item*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

itemsize

Length of one array element in bytes.

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

max (*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)

Return the maximum along a given axis.

Refer to *numpy.amax* for full documentation.

numpy.amax : equivalent function

mean (*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, ***, *where=True*)

Returns the average of the array elements along given axis.

Refer to *numpy.mean* for full documentation.

numpy.mean : equivalent function

min (*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)
Return the minimum along a given axis.

Refer to *numpy.amin* for full documentation.

numpy.amin : equivalent function

nbytes
Total bytes consumed by the elements of the array.

Does not include memory consumed by non-element attributes of the array object.

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

ndim
Number of array dimensions.

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

newbyteorder (*new_order='S'*, */*)
Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

new_order [string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'little'} - little endian
- {'>', 'big'} - big endian
- '=' - native order, equivalent to *sys.byteorder*
- {'l', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order.

new_arr [array] New array object with the dtype reflecting given change to the byte order.

nonzero ()
Return the indices of the elements that are non-zero.

Refer to *numpy.nonzero* for full documentation.

numpy.nonzero : equivalent function

partition(*kth*, *axis=- 1*, *kind='introselect'*, *order=None*)

Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

kth [int or sequence of ints] Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

axis [int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind [{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

order [str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.partition : Return a parititioned copy of an array. argpartition : Indirect partition. sort : Full sort.

See `np.partition` for notes on the different algorithms.

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

prod(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=1*, *where=True*)

Return the product of the array elements over the given axis

Refer to *numpy.prod* for full documentation.

numpy.prod : equivalent function

ptp(*axis=None*, *out=None*, *keepdims=False*)

Peak to peak (maximum - minimum) value along a given axis.

Refer to *numpy.ptp* for full documentation.

numpy.ptp : equivalent function

put(*indices*, *values*, *mode='raise'*)

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *numpy.put* for full documentation.

numpy.put : equivalent function

ravel([*order*])

Return a flattened array.

Refer to *numpy.ravel* for full documentation.

numpy.ravel : equivalent function

ndarray.flat : a flat iterator on the array.

real
The real part of the array.

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

numpy.real : equivalent function

repeat(*repeats*, *axis=None*)
Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

numpy.repeat : equivalent function

reshape(*shape*, *order='C'*)
Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

numpy.reshape : equivalent function

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

resize(*new_shape*, *refcheck=True*)
Change shape and size of array in-place.

new_shape [tuple of ints, or *n* ints] Shape of resized array.

refcheck [bool, optional] If False, reference count will not be checked. Default is True.

None

ValueError If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

SystemError If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

resize : Return a new array with the specified shape.

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing...

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

round(*decimals=0*, *out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to *numpy.around* for full documentation.

numpy.around : equivalent function

searchsorted(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see *numpy.searchsorted*

numpy.searchsorted : equivalent function

setfield(*val*, *dtype*, *offset=0*)

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *dtype* and beginning *offset* bytes into the field.

val [object] Value to be placed in field.

dtype [dtype object] Data-type of the field in which to place *val*.

offset [int, optional] The number of bytes into the field at which to place *val*.

None

getfield

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
```

(continues on next page)

(continued from previous page)

```
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

setflags (*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

write [bool, optional] Describes whether or not *a* can be written to.

align [bool, optional] Describes whether or not *a* is aligned properly for its type.

uic [bool, optional] Describes whether or not *a* is a copy of another "base" array.

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
```

(continues on next page)

(continued from previous page)

```
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

shape

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *numpy.reshape*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[::2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

numpy.reshape : similar function ndarray.reshape : similar method

size

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods

of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

sort (*axis=- 1*, *kind=None*, *order=None*)

Sort an array in-place. Refer to *numpy.sort* for full documentation.

axis [int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

kind [{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0.: The 'stable' option was added.

order [str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

numpy.sort : Return a sorted copy of an array. numpy.argsort : Indirect sort. numpy.lexsort : Indirect stable sort on multiple keys. numpy.searchsorted : Find elements in sorted array. numpy.partition: Partial sort.

See *numpy.sort* for notes on the different sorting algorithms.

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

squeeze (*axis=None*)

Remove axes of length one from *a*.

Refer to *numpy.squeeze* for full documentation.

numpy.squeeze : equivalent function

std (*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, *, *where=True*)

Returns the standard deviation of the array elements along given axis.

Refer to *numpy.std* for full documentation.

numpy.std : equivalent function

strides

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

numpy.lib.stride_tricks.as_strided

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

sum(*axis=None*, *out=None*)

overwrite np.sum(NDVarArray) as people might use it

does not actually support axis/out... todo?

swapaxes(*axis1*, *axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

numpy.swapaxes : equivalent function

take(*indices*, *axis=None*, *out=None*, *mode='raise'*)
Return an array formed from the elements of *a* at the given indices.

Refer to *numpy.take* for full documentation.

numpy.take : equivalent function

tobytes(*order='C'*)
Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object is produced in C-order by default. This behavior is controlled by the `order` parameter.

New in version 1.9.0.

order [{'C', 'F', 'A'}, optional] Controls the memory layout of the bytes object. 'C' means C-order, 'F' means F-order, 'A' (short for *Any*) means 'F' if *a* is Fortran contiguous, 'C' otherwise. Default is 'C'.

s [bytes] Python bytes exhibiting a copy of *a*'s raw data.

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

tofile(*fid*, *sep=''*, *format='%s'*)
Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

fid [file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: *pathlib.Path* objects are now accepted.

sep [str] Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

format [str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

tolist()
Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the *~numpy.ndarray.item* function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

none

y [object, or list of object, or list of list of object, or ...] The possibly nested list of array elements.

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
  ...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

tostring(*order='C'*)

A compatibility alias for *tobytes*, with exactly the same behavior.

Despite its name, it returns *bytes* not *str*s.

Deprecated since version 1.19.0.

trace(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to *numpy.trace* for full documentation.

numpy.trace : equivalent function

transpose(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided

and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

axes : None, tuple of ints, or *n* ints

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

out [ndarray] View of *a*, with axes suitably permuted.

ndarray.T : Array property returning the array transposed. ndarray.reshape : Give a new shape to an array without changing its data.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

var(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*, ***, *where=True*)
Returns the variance of the array elements, along given axis.

Refer to *numpy.var* for full documentation.

numpy.var : equivalent function

view(*[dtype][, type]*)
New view of array with the same data.

Note: Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

---

dtype [data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

type [Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

class cpmpy.variables.**NegBoolView**(*bv*)

Represents not(*var*), not an actual variable implementation!

It stores a link to *var*'s BoolVarImpl

class cpmpy.variables.**NumVarImpl**(*lb*, *ub*)

2.6 Solver Interfaces (cpmpy.solver_interface)

2.6.1 List of classes

SolverInterface	Abstract class for defining solver interfaces.
SolverStatus	Status and statistics of a solver run
ExitStatus	Exit status of the solver

2.6.2 Module description

Contains the abstract class *SolverInterface* for defining solver interfaces, as well as a class *SolverStatus* that collects solver statistics, and the *ExitStatus* class that represents possible exist statuses.

Each solver has its own class that inherits from *SolverInterface*.

class cpmpy.solver_interfaces.solver_interface.**ExitStatus**(*value*)

Exit status of the solver

Attributes: NOT_RUN: Has not been run OPTIMAL: Optimal solution to an optimisation problem found FEASIBLE: Feasible solution to a satisfaction problem found,

or feasible (but not proven optimal) solution to an optimisation problem found

UNSATISFIABLE: No satisfying solution exists ERROR: Some error occured (solver should have thrown Exception)

class cpmpy.solver_interfaces.solver_interface.**SolverInterface**

Abstract class for defining solver interfaces. All classes implementing the SolverInterface

abstract solve(*model*)

Build the CPMpy model into solver-supported model ready for solving and returns the solver statistics generated during model solving.

Parameters **model** (Model) – CPMpy model to be parsed.

Returns an object of *SolverStatus*

abstract supported()

Check for support in current system setup. Return True if the system has package installed or supports solver, else returns False.

Returns: [bool]: Solver support by current system setup.

class cpmpy.solver_interfaces.solver_interface.**SolverStatus**

Status and statistics of a solver run

CHAPTER

THREE

SUPPLEMENTARY `EXAMPLES` PACKAGE

CHAPTER

FOUR

FAQ

Problem: I get the following error:

Solution: Indexing an array with a variable is not allowed by standard numpy arrays, but it is allowed by cpmpy-numpy arrays. First convert your numpy array to a cpmpy-numpy array with the *cparray()* wrapper:

CHAPTER

FIVE

LICENSE

This library is delivered under the MIT License, (see [LICENSE](https://github.com/tias/cppy/blob/master/LICENSE)).

PYTHON MODULE INDEX

C

INDEX

N

P

R

S

T

V