The numpy.ma module

Rationale

Masked arrays are arrays that may have missing or invalid entries. The numpy.ma module provides a nearly work-alike replacement for numpy that supports data arrays with masks.

What is a masked array?

In many circumstances, datasets can be incomplete or tainted by the presence of invalid data. For example, a sensor may have failed to record a data, or recorded an invalid value. The numpy.ma module provides a convenient way to address this issue, by introducing masked arrays.

A masked array is the combination of a standard numpy.ndarray and a mask. A mask is either nomask, indicating that no value of the associated array is invalid, or an array of booleans that determines for each element of the associated array whether the value is valid or not. When an element of the mask is False, the corresponding element of the associated array is valid and is said to be unmasked. When an element of the mask is True, the corresponding element of the associated array is said to be masked (invalid).

The package ensures that masked entries are not used in computations.

As an illustration, let's consider the following dataset:

```
>>> import numpy as np
>>> import numpy.ma as ma
>>> x = np.array([1, 2, 3, -1, 5])
```

We wish to mark the fourth entry as invalid. The easiest is to create a masked array:

```
>>> mx = ma.masked_array(x, mask=[0, 0, 0, 1, 0])
```

We can now compute the mean of the dataset, without taking the invalid data into account:

```
>>> mx.mean()
2.75
```

The numpy.ma module

The main feature of the numpy.ma module is the MaskedArray class, which is a subclass of numpy.ndarray. The class, its attributes and methods are described in more details in the MaskedArray class section.

The numpy.ma module can be used as an addition to numpy:

```
>>> import numpy as np
>>> import numpy.ma as ma
```

To create an array with the second element invalid, we would do:

```
>>> y = ma.array([1, 2, 3], mask = [0, 1, 0])
```

To create a masked array where all values close to 1.e20 are invalid, we would do:

```
>>> z = ma.masked_values([1.0, 1.e20, 3.0, 4.0], 1.e20)
```

For a complete discussion of creation methods for masked arrays please see section Constructing masked arrays.

Using numpy.ma

Constructing masked arrays

There are several ways to construct a masked array.

- A first possibility is to directly invoke the `MaskedArray` class.
- A second possibility is to use the two masked array constructors, `array` and `masked_array`.

`array`(data[, dtype, copy, order, mask, ...])	An array class with possibly masked values.
`masked_array`	alias of `numpy.ma.core.MaskedArray`

- A third option is to take the view of an existing array. In that case, the mask of the view is set to `nomask` if the array has no named fields, or an array of boolean with the same structure as the array otherwise.

```
>>> x = np.array([1, 2, 3])
>>> x.view(ma.MaskedArray)
masked_array(data=[1, 2, 3],
             mask=False,
       fill_value=999999)
>>> x = np.array([(1, 1.), (2, 2.)], dtype=[('a',int), ('b', float)])
>>> x.view(ma.MaskedArray)
masked_array(data=[(1, 1.0), (2, 2.0)],
             mask=[(False, False), (False, False)],
       fill_value=(999999, 1.e+20),
            dtype=[('a', '<i8'), ('b', '<f8')])
```

- Yet another possibility is to use any of the following functions:

`asarray`(a[, dtype, order])	Convert the input to a masked array of the given data-type.
`asanyarray`(a[, dtype])	Convert the input to a masked array, conserving subclasses.
`fix_invalid`(a[, mask, copy, fill_value])	Return input with invalid data masked and replaced by a fill value.
`masked_equal`(x, value[, copy])	Mask an array where equal to a given value.
`masked_greater`(x, value[, copy])	Mask an array where greater than a given value.

masked_greater_equal(x, value[, copy])	Mask an array where greater than or equal to a given value.
masked_inside(x, v1, v2[, copy])	Mask an array inside a given interval.
masked_invalid(a[, copy])	Mask an array where invalid values occur (NaNs or infs).
masked_less(x, value[, copy])	Mask an array where less than a given value.
masked_less_equal(x, value[, copy])	Mask an array where less than or equal to a given value.
masked_not_equal(x, value[, copy])	Mask an array where *not* equal to a given value.
masked_object(x, value[, copy, shrink])	Mask the array *x* where the data are exactly equal to value.
masked_outside(x, v1, v2[, copy])	Mask an array outside a given interval.
masked_values(x, value[, rtol, atol, copy, ...])	Mask using floating point equality.
masked_where(condition, a[, copy])	Mask an array where a condition is met.

Accessing the data

The underlying data of a masked array can be accessed in several ways:

- through the **data** attribute. The output is a view of the array as a **numpy.ndarray** or one of its subclasses, depending on the type of the underlying data at the masked array creation.
- through the **__array__** method. The output is then a **numpy.ndarray**.
- by directly taking a view of the masked array as a **numpy.ndarray** or one of its subclass (which is actually what using the **data** attribute does).
- by using the **getdata** function.

None of these methods is completely satisfactory if some entries have been marked as invalid. As a general rule, where a representation of the array is required without any masked entries, it is recommended to fill the array with the **filled** method.

Accessing the mask

The mask of a masked array is accessible through its **mask** attribute. We must keep in mind that a `True` entry in the mask indicates an *invalid* data.

Another possibility is to use the **getmask** and **getmaskarray** functions. `getmask(x)` outputs the mask of `x` if `x` is a masked array, and the special value **nomask** otherwise. `getmaskarray(x)` outputs the mask of `x` if `x` is a masked array. If `x` has no invalid entry or is not a masked array, the function outputs a boolean array of `False` with as many elements as `x`.

Accessing only the valid entries

To retrieve only the valid entries, we can use the inverse of the mask as an index. The inverse of the mask can be calculated with the **numpy.logical_not** function or simply with the `~` operator:

```
>>> x = ma.array([[1, 2], [3, 4]], mask=[[0, 1], [1, 0]])
>>> x[~x.mask]
masked_array(data=[1, 4],
             mask=[False, False],
       fill_value=999999)
```

Another way to retrieve the valid data is to use the **compressed** method, which returns a one-dimensional **ndarray** (or one of its subclasses, depending on the value of the **baseclass** attribute):

```
>>> x.compressed()
array([1, 4])
```

Note that the output of **compressed** is always 1D.

Modifying the mask

Masking an entry

The recommended way to mark one or several specific entries of a masked array as invalid is to assign the special value **masked** to them:

```
>>> x = ma.array([1, 2, 3])
>>> x[0] = ma.masked
>>> x
masked_array(data=[--, 2, 3],
             mask=[ True, False, False],
       fill_value=999999)
>>> y = ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> y[(0, 1, 2), (1, 2, 0)] = ma.masked
>>> y
masked_array(
  data=[[1, --, 3],
        [4, 5, --],
        [--, 8, 9]],
  mask=[[False,  True, False],
        [False, False,  True],
        [ True, False, False]],
  fill_value=999999)
>>> z = ma.array([1, 2, 3, 4])
>>> z[:-2] = ma.masked
>>> z
masked_array(data=[--, --, 3, 4],
             mask=[ True,  True, False, False],
       fill_value=999999)
```

A second possibility is to modify the **mask** directly, but this usage is discouraged.

> **Note**
>
> When creating a new masked array with a simple, non-structured datatype, the mask is initially set to the special value **nomask**, that corresponds roughly to the boolean `False`. Trying to set an element of **nomask** will fail with a **TypeError** exception, as a boolean does not support item assignment.

All the entries of an array can be masked at once by assigning `True` to the mask:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x.mask = True
>>> x
masked_array(data=[--, --, --],
             mask=[ True,  True,  True],
       fill_value=999999,
            dtype=int64)
```

Finally, specific entries can be masked and/or unmasked by assigning to the mask a sequence of booleans:

```
>>> x = ma.array([1, 2, 3])
>>> x.mask = [0, 1, 0]
>>> x
masked_array(data=[1, --, 3],
             mask=[False,  True, False],
       fill_value=999999)
```

Unmasking an entry

To unmask one or several specific entries, we can just assign one or several new valid values to them:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x[-1] = 5
>>> x
masked_array(data=[1, 2, 5],
             mask=[False, False, False],
       fill_value=999999)
```

Note

Unmasking an entry by direct assignment will silently fail if the masked array has a *hard* mask, as shown by the **hardmask** attribute. This feature was introduced to prevent overwriting the mask. To force the unmasking of an entry where the array has a hard mask, the mask must first to be softened using the **soften_mask** method before the allocation. It can be re-hardened with **harden_mask**:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1], hard_mask=True)
>>> x
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x[-1] = 5
>>> x
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x.soften_mask()
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x[-1] = 5
>>> x
masked_array(data=[1, 2, 5],
             mask=[False, False, False],
       fill_value=999999)
>>> x.harden_mask()
masked_array(data=[1, 2, 5],
             mask=[False, False, False],
       fill_value=999999)
```

To unmask all masked entries of a masked array (provided the mask isn't a hard mask), the simplest solution is to assign the constant **nomask** to the mask:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x.mask = ma.nomask
>>> x
masked_array(data=[1, 2, 3],
             mask=[False, False, False],
       fill_value=999999)
```

Indexing and slicing

As a **MaskedArray** is a subclass of **numpy.ndarray**, it inherits its mechanisms for indexing and slicing.

When accessing a single entry of a masked array with no named fields, the output is either a scalar (if the corresponding entry of the mask is `False`) or the special value **masked** (if the corresponding entry of the mask is `True`):

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x[0]
1
>>> x[-1]
masked
>>> x[-1] is ma.masked
True
```

If the masked array has named fields, accessing a single entry returns a **numpy.void** object if none of the fields are masked, or a 0d masked array with the same dtype as the initial array if at least one of the fields is masked.

```
>>> y = ma.masked_array([(1,2), (3, 4)],
...                     mask=[(0, 0), (0, 1)],
...                    dtype=[('a', int), ('b', int)])
>>> y[0]
(1, 2)
>>> y[-1]
(3, --)
```

When accessing a slice, the output is a masked array whose **data** attribute is a view of the original data, and whose mask is either **nomask** (if there was no invalid entries in the original array) or a view of the corresponding slice of the original mask. The view is required to ensure propagation of any modification of the mask to the original.

```
>>> x = ma.array([1, 2, 3, 4, 5], mask=[0, 1, 0, 0, 1])
>>> mx = x[:3]
>>> mx
masked_array(data=[1, --, 3],
             mask=[False,  True, False],
       fill_value=999999)
>>> mx[1] = -1
>>> mx
masked_array(data=[1, -1, 3],
             mask=[False, False, False],
       fill_value=999999)
>>> x.mask
array([False, False, False, False,  True])
>>> x.data
array([ 1, -1,  3,  4,  5])
```

Accessing a field of a masked array with structured datatype returns a **MaskedArray**.

Operations on masked arrays

Arithmetic and comparison operations are supported by masked arrays. As much as possible, invalid entries of a masked array are not processed, meaning that the corresponding **data** entries *should* be the same before and after the operation.

> **Warning**
>
> We need to stress that this behavior may not be systematic, that masked data may be affected by the operation in some cases and therefore users should not rely on this data remaining unchanged.

The **numpy.ma** module comes with a specific implementation of most ufuncs. Unary and binary functions that have a validity domain (such as **log** or **divide**) return the **masked** constant whenever the input is masked or falls outside the validity domain:

```
>>> ma.log([-1, 0, 1, 2])
masked_array(data=[--, --, 0.0, 0.6931471805599453],
             mask=[ True,  True, False, False],
       fill_value=1e+20)
```

Masked arrays also support standard numpy ufuncs. The output is then a masked array. The result of a unary ufunc is masked wherever the input is masked. The result of a binary ufunc is masked wherever any of the input is masked. If the ufunc also returns the optional context output (a 3-element tuple containing the name of the ufunc, its arguments and its domain), the context is processed and entries of the output masked array are masked wherever the corresponding input fall outside the validity domain:

```
>>> x = ma.array([-1, 1, 0, 2, 3], mask=[0, 0, 0, 0, 1])
>>> np.log(x)
masked_array(data=[--, 0.0, --, 0.6931471805599453, --],
             mask=[ True, False,  True, False,  True],
       fill_value=1e+20)
```

Examples

Data with a given value representing missing data

Let's consider a list of elements, x, where values of -9999. represent missing data. We wish to compute the average value of the data and the vector of anomalies (deviations from the average):

```
>>> import numpy.ma as ma
>>> x = [0.,1.,-9999.,3.,4.]
>>> mx = ma.masked_values (x, -9999.)
>>> print(mx.mean())
2.0
>>> print(mx - mx.mean())
[-2.0 -1.0 -- 1.0 2.0]
>>> print(mx.anom())
[-2.0 -1.0 -- 1.0 2.0]
```

Filling in the missing data

Suppose now that we wish to print that same data, but with the missing values replaced by the average value.

```
>>> print(mx.filled(mx.mean()))
[ 0.  1.  2.  3.  4.]
```

Numerical operations

Numerical operations can be easily performed without worrying about missing values, dividing by zero, square roots of negative numbers, etc.:

```
>>> import numpy.ma as ma
>>> x = ma.array([1., -1., 3., 4., 5., 6.], mask=[0,0,0,0,1,0])
>>> y = ma.array([1., 2., 0., 4., 5., 6.], mask=[0,0,0,0,0,1])
>>> print(ma.sqrt(x/y))
[1.0 -- -- 1.0 -- --]
```

Four values of the output are invalid: the first one comes from taking the square root of a negative number, the second from the division by zero, and the last two where the inputs were masked.

Ignoring extreme values

Let's consider an array `d` of floats between 0 and 1. We wish to compute the average of the values of `d` while ignoring any data outside the range `[0.2, 0.9]`:

```
>>> d = np.linspace(0, 1, 20)
>>> print(d.mean() - ma.masked_outside(d, 0.2, 0.9).mean())
-0.05263157894736836
```

<< Masked arrays

numpy.ma.array >>

Last updated on Jan 31, 2021.
Created using Sphinx 2.4.4.