Tutorial-Coercions_and_actions

Implementing arithmetic operations in Sage

Coercions and actions

© Simon King (September 2013)
Friedrich-Schiller-Universität Jena
simon.king@uni-jena.de

The aim of this tutorial is to show what needs to be implemented in order to enable arithmetic operations between new and existing algebraic structures in Sage.

Table of contents

In this tutorial, we are implementing the set of m by n matrices over a given commutative ring, subject to the usual multiplication rules of matrices. Our implementation is very slow and naive, and it will of course not replace the existing implementation of matrix spaces. Our toy implementation only aims to demonstrate how one can implement

1. Multiplication within an algebra (square matrices)
2. Mixing "new" and "old" matrices in arithmetics
3. Multiplication of rectangular matrices
4. Action by, e.g., permutation groups (and similarly action on, say, polynomials)

Note that the existing implementation of matrix spaces in Sage does not implement coercion in the same way as demonstrated here.

1. Arithmetics within one parent

This is not a talk about Sage's category framework, but we will need "categorial notions" in various places.

Def:

A *Parent* is an object of a sub-category of the category of sets. In particular, it contains elements.

Sage provides convenient base classes for a number of algebraic parent structures and their elements, and they should be used.

In addition, Sage provides a "category framework", and one should indicate the category which a parent belongs to---*in addition to* using algebraic base classes!

Matrix spaces are modules, and they are algebras if the matrices are square. Hence, we build our matrix spaces on top of a base class sage.modules.module.Module, and refine the class using the category framework in the case of square matrices.

It may seem strange that we build non-square matrices on top of a base class for *ring* elements, but the base class of *module* elements strictly disallows n internal multiplication (which I think is a bug).

The following tentative class definitions will be refined later.

```
from sage.modules.module import Module
from sage.structure.parent import Parent
class MyMatrixSpace(Module):
    def __init__(self, B, m, n=None):
        # the base ring must be commutative
        if B not in CommutativeRings():
            raise ValueError, "commutative ring required"
        if n is None:
            n = m
        self.m = m
        self.n = n
        # Module.__init__ should accept an argument "category"
and pass it through,
        # but it currently doesn't <BUG!>
        # Anyway, Module.__init__ does not more than calling
Parent.__init__, so, the following is OK:
        if m==n:
            if m==1:
                Parent.__init__(self, base=B,
category=CommutativeAlgebras(B))
            else:
                Parent.__init__(self, base=B,
category=Algebras(B))
        else:
            Parent.__init__(self, base=B, category=Modules(B))
    def nrows(self):
        return self.m
    def ncols(self):
        return self.n
    def _repr_(self):
        # NOTE: Single underscore
        return "Our space of %sx%s matrices over %s"%
(self.m,self.n,self.base())
    def __cmp__(self, other):
        c = cmp(self.__class__, other.__class__)
        if c:
            return c
        c = cmp(self.nrows(), other.nrows())
        if c:
            return c
        c = cmp(self.ncols(), other.ncols())
        if c:
            return c
        return cmp(self.base(), other.base())
```

Sometimes our matrix spaces are just modules and not algebras. However, Sage's base class <ModuleElements> raises an error if an internal multiplication is attempted. Hence, we start with <RingElement> and square matrices, and cope with the absence of internal multiplication for non-square matrices later. First, we only take care of the data structure and the string representation:

```
from sage.rings.ring_element import RingElement
class MyMatrix(RingElement):
    def __init__(self, parent, data):
        self.data = [list(r) for r in data]
        RingElement.__init__(self, parent)
    def _repr_(self):
        return os.linesep.join([repr(r) for r in self.data])
    def __iter__(self):
        for r in self.data:
            yield r
    def __getitem__(self, key):
        try:
            return self.data[key]
        except TypeError:
            if len(key)==2:
                if isinstance(key[0], slice):
                    return [row[key[1]] for row in
self.data[key[0]]]
                return self.data[key[0]][key[1]]
            raise ValueError, "invalid indices"
```

The **first lesson** of this lecture is: Using the category framework to create elements is easy!

1. Assign a class to an attribute "Element" of the parent (or of the parent's class)
2. The parent will then provide an attribute "element_class".
3. The default way of creating elements will immediately work, and yields instances of the "element_class".

```
MyMatrixSpace.Element = MyMatrix
```

```
MS = MyMatrixSpace(QQ, 2)
```

```
MS.element_class; MS.element_class==MyMatrix;
issubclass(MS.element_class, MyMatrix)
```

```
<class '__main__.MyMatrixSpace_with_category.element_class'>
False
True
```

```
M1 = MS([[1,2],[3,4]])
```

```
M1; isinstance(M1, MS.element_class); M1.parent() is MS
```

```
[1, 2]
[3, 4]
True
True
```

Slicing works (we will use it in the multiplication code below):

```
M1[:,1]
```

```
[2, 4]
```

```
M1[0,:]
```

```
[1, 2]
```

We now come to the arithmetic methods. You certainly know about Python's "magical" double underscore methods. Usually, the base classes in Sage have default implementations. Here is the default multiplication code for ring elements (I have removed the long doc string from the output):

```
from sage.misc.sageinspect import sage_getsourcelines
sources = sage_getsourcelines(RingElement.__mul__)[0]
pretty_print(''.join([sources[0]]+sources[-10:]))
```

```
    def __mul__(left, right):
        if have_same_parent(left, right):
            if  (<RefPyObject *>left).ob_refcnt < inplac
                return (<RingElement>left)._imul_(<RingE
            else:
                return (<RingElement>left)._mul_(<RingEl
        if PyInt_CheckExact(right):
            return (<ModuleElement>left)._mul_long(PyInt
        elif PyInt_CheckExact(left):
            return (<ModuleElement>right)._mul_long(PyIn
        return coercion_model.bin_op(left, right, mul)
```

The **second lesson** of this lecture is: Do not override __mul__ (nor __add__, __sub__, __call__) unless you are ready to bear the consequences of what you are doing, since the default implementations are essential for Sage's coercion framework!

Generally, you should provide the following ***single underscore*** methods:

1. _repr_ --- for string representation
2. _mul_, _add_, _sub_ --- for multiplication, addition, subtraction within one parent (_sub_ is optional)
3. _rmul_ / _lmul_ --- for multiplication by base ring elements from the left/right

Sage's coercion model guarantees that

1. both arguments of _mul_, _add_ and _sub_ belong to ***the same parent***
2. the second argument of _rmul_ and _lmul_ belongs to the base ring

```
class MyMatrix(MyMatrix):  # Yes, Python allows it. I don't want
to copy-and-paste the whole code!
    def _add_(self, other):
        # we can assume that self and other have the same
parent!
        P = self.parent()
        return P([[a+b for a,b in zip(r1,r2)] for r1,r2 in
zip(self,other)])
    def _mul_(self, other):
        # we can assume that self and other have the same
parent!
        P = self.parent()
        if P.nrows()!=P.ncols():
            raise ValueError, "can't multiply within non-square
matrix spaces"
        return P([[add([a*b for a,b in zip(self[m],
other[:,k])])
                                  for k in range(P.ncols())]
                                      for m in range(P.nrows())])
```

Recreate the matrix space and define some elements:

```
MyMatrixSpace.Element = MyMatrix
MS = MyMatrixSpace(QQ,2)
M1 = MS([[1,2],[3,4]])
M2 = MS([[0,1],[2,1]])
```

Addition works, due to _add_:

```
M1 + M2
```

```
    [1, 3]
    [5, 5]
```

Square matrix multiplication in _mul_ works as well:

```
M1*M2
```

```
    [4, 3]
    [8, 7]
```

Verify this multiplication with Sage's matrices:

```
m1 = matrix([[1,2],[3,4]])
m2 = matrix([[0,1],[2,1]])
m1*m2
```

```
    [4 3]
    [8 7]
```

Multiplication by base ring elements does not work, yet.

```
M1*2
```

```
    Traceback (click to the left of this block for traceback)
    ...
    TypeError: unsupported operand parent(s) for '*': 'Our space of
    matrices over Rational Field' and 'Integer Ring'
```

We fix it by providing _lmul_ and _rmul_.

```
class MyMatrix(MyMatrix):
    def _lmul_(self, c):
        return self.parent()([[c*a for a in row] for row in
self])
    def _rmul_(self, c):
        return self.parent()([[a*c for a in row] for row in
self])
```

```
MyMatrixSpace.Element = MyMatrix
MS = MyMatrixSpace(QQ,2)
M1 = MS([[1,2],[3,4]])
M2 = MS([[0,1],[2,1]])
```

```
M1*2
```

```
    [2, 4]
    [6, 8]
```

```
3*M2
```

```
    [0, 3]
    [6, 3]
```

2. Conversions and coercions between two parents

Conversion

Generally, a conversion is a procedure that turns given data into an element of a parent.

Normally, one wants to convert elements of the base ring into diagonal matrices. This is not implemented, yet:

```
MS(1)
```

```
    Traceback (click to the left of this block for traceback)
    ...
    TypeError: 'int' object is not iterable
```

We could fix this by making MyMatrix.__init__ accept a base ring element as input argument. However, the preferred way is to implement a method **_element_constructor_** for the parent.

The **third lesson** of this lecture is: Do not override the default __call__ method, but implement _element_constructor_!!

```
class MyMatrixSpace(MyMatrixSpace):
    def _element_constructor_(self, data):
        B = self.base()
        z = B.zero()
        if data is None:
            data = z
        if data in B:
            return self.element_class(self, [[z]*k+[data]+[z]*
(self.n-k-1) for k in range(self.m)])
        return self.element_class(self, data)
```

```
MS = MyMatrixSpace(QQ,2)
```

Conversions of lists of lists still work:

```
M1 = MS([[1,2],[3,4]])
M2 = MS([[0,1],[2,1]])
M1; isinstance(M1, MS.element_class)
```

```
    [1, 2]
    [3, 4]
    True
```

In addition, we can convert elements of the base ring intor diagonal matrices:

```
MS(2)
```

```
    [2, 0]
    [0, 2]
```

We can also convert the usual Sage matrices into our matrix space (this is because MyMatrix.__init__ expects arguments that are iterables of iterables):

```
SageMatQQ = matrix(QQ, [[1,2],[3,4]])
MS(SageMatQQ)
```

```
    [1, 2]
    [3, 4]
```

Coercions

Of course, it would be a very bad idea to automatically apply an arbitrary conversions when doing arithmetic. Sage would by default not apply the conversion of SageMatQQ into MS for doing arithmetic.

Hence, currently the addition of M1 and SageMatQQ fails, even though it would "mathematically reasonably" work:

```
M1+SageMatQQ
```

```
    Traceback (click to the left of this block for traceback)
    ...
    TypeError: unsupported operand parent(s) for '+': 'Our space of
    matrices over Rational Field' and 'Full MatrixSpace of 2 by 2 d
    matrices over Rational Field'
```

Why would this addition be "mathematically reasonable"? This is related with the notion of a **coercion**.

Def:

Let P_1, P_2, P_3 be parents.

1. A **coercion** ϕ from P_1 to P_2 is a morphism in the meet $\mathcal{C}$ of the categories of P_1 and P_2:

   $\phi \in \mathrm{Hom}_{\mathcal{C}}(P_1, P_2)$

2. There is *at most one coercion* from one parent to another, and if it exists, then it coincides with conversion.

3. The identity morphism is a coercion.

4. If ϕ is a coercion from P_1 to P_2 and ψ is a coercion from P_2 to P_3, then $\psi \circ \phi$ is a (the!) coercion from P_1 to P_3

Coercions will implicitly be used in arithmetic operations!

Warning/Example:

If P_1 is an additive group and P_2 is a ring, and P_1 coerces into P_2 by a morphism in the category of additive groups, then this coercion will also be used to ***multiply*** elements of P_1 with elements of P_2 (the result lives in P_2)!

By now, Sage knows that there is a conversion from Sage matrices into our matrix space, but it doesn't know that it is a coercion:

```
MS.convert_map_from(SageMatQQ.parent())
```

```
    Conversion map:
      From: Full MatrixSpace of 2 by 2 dense matrices over Rational
    Field
      To:   Our space of 2x2 matrices over Rational Field
```

```
MS.has_coerce_map_from(SageMatQQ.parent())
```

```
    False
```

In contrast, Sage knows that the base ring of MS coerces into MS. This is automatically done for any parent that is initialised in the category of algebras over a base ring: We see the category framework at work.

```
MS.coerce_map_from(ZZ)
```

```
    Composite map:
      From: Integer Ring
      To:   Our space of 2x2 matrices over Rational Field
      Defn:   Natural morphism:
                From: Integer Ring
                To:   Rational Field
              then
                Generic morphism:
                From: Rational Field
                To:   Our space of 2x2 matrices over Rational Field
```

Hence, we can already add an integer to a square matrix: The integer is implicitly converted into a diagonal matrix.

```
1+M1
```

```
    [2, 2]
    [3, 5]
```

In order to make Sage aware of a coercion, one can implement a _coerce_map_from_() method. If it returns True, then an existing conversion is used as coercion. If it returns a map, then this map is used for coercion.

```
class MyMatrixSpace(MyMatrixSpace):
    def _coerce_map_from_(self, P):
        # Coercion from the base ring has already been taken
care of.
        # We want coercion from other matrix spaces of
compatible parameters.
        # Use duck-typing!
        try:
            return self.m==P.nrows() and self.n==P.ncols() and
self.base().has_coerce_map_from(P.base())
        except AttributeError:
            return False
```

```
MS = MyMatrixSpace(QQ,2)
M1 = MS([[1,2],[3,4]])
M2 = MS([[0,1],[2,1]])
```

Now, Sage detects coercions from "usual" matrix spaces into "our" matrix space.

```
MSSage = MatrixSpace(ZZ,2)
MS.coerce_map_from(MSSage)
```

```
    Conversion map:
      From: Full MatrixSpace of 2 by 2 dense matrices over Integer
      To:   Our space of 2x2 matrices over Rational Field
```

```
M1 * MSSage(2)   # Note that the result is using "our"
implementation
```

```
    [2, 4]
    [6, 8]
```

```
MSSage(1) + M1
```

```
    [2, 2]
    [3, 5]
```

3. Pushout construction: Automatic creation of new parents (a teaser)

The material of this section is treated in more detail in the thematic tutorial on "coercions and categories".

Many parents in Sage can tell how they are constructed:

```
FSage, R = MSSage.construction()
FSage, R
```

```
    (MatrixFunctor, Integer Ring)
```

```
FSage(R) == MSSage
```

```
    True
```

Here, we see a so-called ***construction functor***. This is a covariant functor $\mathcal{F} : \mathcal{C}_1 \to \mathcal{C}_2$, so that applying $\mathcal{F}$ to any coercion map $\phi \in \mathrm{Hom}_{\mathcal{C}_1}(P, Q)$ yields a coercion map from $\mathcal{F}(P)$ to $\mathcal{F}(Q)$. Examples include:

- the construction of a matrix space
- the construction of a polynomial ring
- the construction of a fraction field
- the construction of the algebraic closure.

```
FP,R = QQ['x'].construction(); FP,R; FP.domain(), FP.codomain()
```

```
    (Poly[x], Rational Field)
    (Category of rings, Category of rings)
```

```
FF,R = QQ.construction(); FF,R; FF.domain(), FF.codomain()
```

```
    (FractionField, Integer Ring)
    (Category of integral domains, Category of fields)
```

```
FC,R = QQbar.construction(); FC,R; FC.domain(), FC.codomain()
    (AlgebraicClosureFunctor, Rational Field)
    (Category of rings, Category of rings)
```

Often, there is a coercion from R to $\mathcal{F}(R)$, for any object R of $\mathcal{C}_1$. Here is an exception of this rule:

```
MatrixSpace(ZZ,2,3).has_coerce_map_from(ZZ)
    False
```

Every construction functor has a "rank":

```
FC.rank, FF.rank, FP.rank, FSage.rank
    (3, 5, 9, 10)
```

If two parents P_1 , P_2 are obtained from a common "ancestor" by two sequences of construction functors, then Sage will try to "shuffle" the two construction sequences, applying first a functor with *lower* rank. The result is called the *pushout* of P_1 and P_2 , and is a reasonable candidate for performing arithmetic operations involving elements from both P_1 and P_2 :

```
FSage(FF(ZZ))
    Full MatrixSpace of 2 by 2 dense matrices over Rational Field
FSage(FP(ZZ))
    Full MatrixSpace of 2 by 2 dense matrices over Univariate Polyn
    Ring in x over Integer Ring
```

```
from sage.categories.pushout import pushout
pushout(FSage(FF(ZZ)), FSage(FP(ZZ)))   # FF has lower rank than
FP and is applied first in the pushout
    Full MatrixSpace of 2 by 2 dense matrices over Univariate Polyn
    Ring in x over Rational Field
```

When two different functors of the same rank clash when shuffling, Sage will raise an error, unless it is asserted that one can apply the functors in any order, or Sage is told how to merge the two functors into one.

Here, we create a new matrix construction functor, imposing the rule that Sage's matrix functors will henceforth merge into *our* matrix functors:

```
from sage.categories.pushout import MatrixFunctor
class MyMatrixFunctor(MatrixFunctor):
    def _apply_functor(self, R):
        return MyMatrixSpace(R,self.nrows,self.ncols)
    def merge(self, other):
        if not isinstance(other, MatrixFunctor):
            return
        if other.nrows!=self.nrows or other.ncols!=self.ncols:
            return
        return self
```

```
class MyMatrixSpace(MyMatrixSpace):
    def construction(self):
        return MyMatrixFunctor(self.m,self.n), self.base()
```

```
MS = MyMatrixSpace(QQ,2)
M1 = MS([[1,2],[3,4]])
M2 = MS([[0,1],[2,1]])
```

```
MyF, R = MS.construction()
MyF(R) == MS
```

```
    True
```

Now, if Sage can not find a coercion from one parent into another, then it constructs a pushout and coerces into it:

```
MSPolySage = MatrixSpace(ZZ['x'],2)
MS.has_coerce_map_from(MSPolySage),
MSPolySage.has_coerce_map_from(MS)
```

```
    (False, False)
```

```
pushout(MS, MSPolySage)   # this is because merging prefers
*our* implementation!
```

```
    Our space of 2x2 matrices over Univariate Polynomial Ring in x
    Rational Field
```

```
MSPolySage('x')
```

```
    [x 0]
    [0 x]
```

```
MSPolySage('x') + MS(1/2)
```

```
    [x + 1/2, 0]
    [0, x + 1/2]
```

4. Actions

"Act-on" and "acted-upon" actions

One way of implementing an action of anything on an element is: Implement a method _acted_upon_ of the element.

```
RingElement._acted_upon_??
```

File: /home/king/Sage/prerelease/sage-5.12.beta5/devel/sage/sage/structure/element.pyx

Source Code (starting at line 716):

```
cpdef _acted_upon_(self, x, bint self_on_left):
    """
    Use this method to implement self acted on by x.

    Return None or raise a CoercionException if no
    such action is defined here.
    """
    return None
```

Similarly, an action of the element on anything can be defined by a method _act_on_ of the element.

We implement a left action of a permutation group on our matrices, by permuting rows.

```
class MyMatrix(MyMatrix):
    def _acted_upon_(self, x, self_on_left):
        if self_on_left:
            return None
        # Here, we mainly use that permutations are callable
        return self.parent()([self.data[x(i)] for i in
range(self.parent().nrows())])
```

Refresh our matrix space definition, and let permutations act:

```
MyMatrixSpace.Element = MyMatrix
MS = MyMatrixSpace(QQ,4)
M1 = MS(1)
M2 = MS([[1,2,3,4], [5,6,7,8], [8,7,6,5], [4,3,2,1]])
```

```
M1
```

```
    [1, 0, 0, 0]
    [0, 1, 0, 0]
    [0, 0, 1, 0]
    [0, 0, 0, 1]
```

```
P = Permutation([1,3,2,4])
```

*@#/}!!!

Someone has killed Sage's category framework for permutations, by overloading __mul__! You should not do that!

```
P * M1
```

```
    Traceback (click to the left of this block for traceback)
    ...
    TypeError: object of type
    'MyMatrixSpace_with_category.element_class' has no len()
```

```
P.__mul__??
```

File: /home/king/Sage/prerelease/sage-5.12.beta5/local/lib/python2.7/site-packages/sage/combinat/permutation.py

Source Code (starting at line 1244):

```
def __mul__(self, rp):
    """
    TESTS::

        sage: p213 = Permutation([2,1,3])
        sage: p312 = Permutation([3,1,2])
        sage: PermutationOptions(mult='l2r')
        sage: p213*p312
        [1, 3, 2]
        sage: PermutationOptions(mult='r2l')
        sage: p213*p312
        [3, 2, 1]
        sage: PermutationOptions(mult='l2r')
    """
    if self.parent().global_options['mult'] == 'l2r':
        return self._left_to_right_multiply_on_right(rp)
    else:
        return self._left_to_right_multiply_on_left(rp)
```

So, we better work with an element of the symmetric group, which complies with Sage's coercion framework:

```
p = SymmetricGroup(4)([(3,2)]); p
```

```
    (2,3)
```

Now, multiplication from the left works, but (on purpose) not from the right:

```
p*M1
```

```
    [1, 0, 0, 0]
    [0, 1, 0, 0]
    [0, 0, 0, 1]
    [0, 0, 1, 0]
```

```
M1*p
```

```
    Traceback (click to the left of this block for traceback)
    ...
    TypeError: unsupported operand parent(s) for '*': 'Our space of
    matrices over Rational Field' and 'Symmetric group of order 4!
    permutation group'
```

Sage assumes that the action of the symmetric group on our matrix space will always use _acted_upon_:

```
A = MS.get_action(p.parent(), self_on_left=False); A; type(A)
```

```
    Left action by Symmetric group of order 4! as a permutation gro
    Our space of 4x4 matrices over Rational Field
    <type 'sage.structure.coerce_actions.ActedUponAction'>
```

Defining arbitrary actions of parents

Sometimes _acted_upon_/_act_on_ on the level of elements is not good enough. It is also possible to let the *parent* take care of creating an action.

Of course, there is a base class for actions. To create a new type of actions, implement a _call_ method (not __call__) that take two elements and returns the result of the action of these elements.

```
from sage.categories.action import Action
class MyMatrixAction(Action):
    def __init__(self, P1,P2):
        Action.__init__(self, P1, P2, is_left=True,
op=operator.mul)
        self.P = MyMatrixSpace(pushout(P1.base(),P2.base()),
P1.nrows(), P2.ncols())
        self.range = P1.ncols()
    def _repr_name_(self):
        # helps for string representation
        return "multiplication action of our matrices"
    def _call_(self, left,right):
        return self.P([[add([a*b for a,b in zip(left[m],
                                                 [right[i,k] for
i in range(self.range)])
                             ])
                             for k in range(self.P.ncols())]
```

```
                                 for m in
range(self.P.nrows())])
```

A parent can use _get_action_ to indicate the action being used:

```
class MyMatrixSpace(MyMatrixSpace):
    def _get_action_(self, P, op, self_on_left):
        if op!=operator.mul:
            return Module._get_action_(self, P, op,
self_on_left)
        try:
            # duck type matrix spaces:
            Pcol = P.ncols()
            Prow = P.nrows()
        except AttributeError:
            return Module._get_action_(self, P, op,
self_on_left)
        if self_on_left:
            if self.ncols()!=Prow:
                return Module._get_action_(self, P, op,
self_on_left)
            else:
                return MyMatrixAction(self,P)
        else:
            if Pcol!=self.nrows():
                return Module._get_action_(self, P, op,
self_on_left)
            else:
                return MyMatrixAction(P, self)
```

```
MS1 = MyMatrixSpace(QQ,2,4); MS2 = MyMatrixSpace(QQ,4,2)
```

```
MS1.get_action(MS2)
```

```
    Left multiplication action of our matrices by Our space of 2x4
    matrices over Rational Field on Our space of 4x2 matrices over
    Rational Field
```

Now, we define matrices both in our and in Sage's implementation, and see that by our definition of a multiplication action, we can multiply the rectangular matrices, even if different implementations are involved:

```
M1 = MS1([[1,2,3,4],[5,6,7,8]]); M2 = MS2([[1,2],[3,4],[5,6],
[7,8]])
m1 = matrix(QQ, [[1,2,3,4],[5,6,7,8]]); m2 = matrix(QQ, [[1,2],
[3,4],[5,6],[7,8]])
```

```
M1*M2
```

```
    [50, 60]
    [114, 140]
```

```
m1*m2    # Check our result against Sage's result
```

```
    [ 50  60]
    [114 140]
```

```
m1*M2    # Sage's matrix times our matrix
```

```
    [50, 60]
    [114, 140]
```

```
M1*m2    # our matrix times Sage's matrix
```

```
    [50, 60]
    [114, 140]
```

And verify if the old stuff still works:

```
2*M1
```

```
    [2, 4, 6, 8]
    [10, 12, 14, 16]
```

```
M2*3
```

```
    [3, 6]
    [9, 12]
    [15, 18]
    [21, 24]
```

```
p*M2
```

```
    [1, 2]
    [3, 4]
    [7, 8]
    [5, 6]
```