```python
#LyX
"""Implementation of :class:`Domain` class. """


from typing import Any, Optional, Type

from sympy.core.numbers import AlgebraicNumber
from sympy.core import Basic, sympify
from sympy.core.sorting import default_sort_key, ordered
from sympy.external.gmpy import HAS_GMPY
from sympy.polys.domains.domainelement import DomainElement
from sympy.polys.orderings import lex
from sympy.polys.polyerrors import UnificationFailed, CoercionFailed, DomainError
from sympy.polys.polyutils import _unify_gens, _not_a_coeff
from sympy.utilities import public
from sympy.utilities.iterables import is_sequence


@public
class Domain:
    """Superclass for all domains in the polys domains system.

    See :ref:`polys-domainsintro` for an introductory explanation of the
    domains system.

    The :py:class:`~.Domain` class is an abstract base class for all of the
    concrete domain types. There are many different :py:class:`~.Domain`
    subclasses each of which has an associated ``dtype`` which is a class
    representing the elements of the domain. The coefficients of a
    :py:class:`~.Poly` are elements of a domain which must be a subclass of
    :py:class:`~.Domain`.
```

```

    Examples
    ========

    The most common example domains are the integers :ref:`ZZ` and the
    rationals :ref:`QQ`.

    >>> from sympy import Poly, symbols, Domain
    >>> x, y = symbols('x, y')
    >>> p = Poly(x**2 + y)
    >>> p
    Poly(x**2 + y, x, y, domain='ZZ')
    >>> p.domain
    ZZ
    >>> isinstance(p.domain, Domain)
    True
    >>> Poly(x**2 + y/2)
    Poly(x**2 + 1/2*y, x, y, domain='QQ')

    The domains can be used directly in which case the domain object e.g.
    (:ref:`ZZ` or :ref:`QQ`) can be used as a constructor for elements of
    ``dtype``.

    >>> from sympy import ZZ, QQ
    >>> ZZ(2)
    2
    >>> ZZ.dtype  # doctest: +SKIP
    <class 'int'>
    >>> type(ZZ(2))  # doctest: +SKIP
    <class 'int'>
    >>> QQ(1, 2)
    1/2
    >>> type(QQ(1, 2))  # doctest: +SKIP
    <class 'sympy.polys.domains.pythonrational.PythonRational'>

    The corresponding domain elements can be used with the arithmetic
```

```
    operations ``+,-,*,**`` and depending on the domain some combination of
    ``/,//,%`` might be usable. For example in :ref:`ZZ` both ``//`` (floor
    division) and ``%`` (modulo division) can be used but ``/`` (true
    division) cannot. Since :ref:`QQ` is a :py:class:`~.Field` its elements
    can be used with ``/`` but ``//`` and ``%`` should not be used. Some
    domains have a :py:meth:`~.Domain.gcd` method.

    >>> ZZ(2) + ZZ(3)
    5
    >>> ZZ(5) // ZZ(2)
    2
    >>> ZZ(5) % ZZ(2)
    1
    >>> QQ(1, 2) / QQ(2, 3)
    3/4
    >>> ZZ.gcd(ZZ(4), ZZ(2))
    2
    >>> QQ.gcd(QQ(2,7), QQ(5,3))
    1/21
    >>> ZZ.is_Field
    False
    >>> QQ.is_Field
    True

    There are also many other domains including:

        1. :ref:`GF(p)` for finite fields of prime order.
        2. :ref:`RR` for real (floating point) numbers.
        3. :ref:`CC` for complex (floating point) numbers.
        4. :ref:`QQ(a)` for algebraic number fields.
        5. :ref:`K[x]` for polynomial rings.
        6. :ref:`K(x)` for rational function fields.
        7. :ref:`EX` for arbitrary expressions.

    Each domain is represented by a domain object and
```

```
    also an implementation
    class (``dtype``) for the elements of the domain. For example the
    :ref:`K[x]` domains are represented by a domain object which is an
    instance of :py:class:`~.PolynomialRing` and the elements are always
    instances of :py:class:`~.PolyElement`. The implementation class
    represents particular types of mathematical expressions in a way that is
    more efficient than a normal SymPy expression which is of type
    :py:class:`~.Expr`. The domain methods :py:meth:`~.Domain.from_sympy` and
    :py:meth:`~.Domain.to_sympy` are used to convert from :py:class:`~.Expr`
    to a domain element and vice versa.

    >>> from sympy import Symbol, ZZ, Expr
    >>> x = Symbol('x')
    >>> K = ZZ[x]           # polynomial ring domain
    >>> K
    ZZ[x]
    >>> type(K)             # class of the domain
    <class 'sympy.polys.domains.polynomialring.PolynomialRing'>
    >>> K.dtype             # class of the elements
    <class 'sympy.polys.rings.PolyElement'>
    >>> p_expr = x**2 + 1   # Expr
    >>> p_expr
    x**2 + 1
    >>> type(p_expr)
    <class 'sympy.core.add.Add'>
    >>> isinstance(p_expr, Expr)
    True
    >>> p_domain = K.from_sympy(p_expr)
    >>> p_domain            # domain element
    x**2 + 1
    >>> type(p_domain)
    <class 'sympy.polys.rings.PolyElement'>
```

```
    >>> K.to_sympy(p_domain) == p_expr
    True
    The :py:meth:`~.Domain.convert_from` method is used to convert domain
    elements from one domain to another.

    >>> from sympy import ZZ, QQ
    >>> ez = ZZ(2)
    >>> eq = QQ.convert_from(ez, ZZ)
    >>> type(ez)  # doctest: +SKIP
    <class 'int'>
    >>> type(eq)  # doctest: +SKIP
    <class 'sympy.polys.domains.pythonrational.PythonRational'>

    Elements from different domains should not be mixed in arithmetic or other
    operations: they should be converted to a common domain first.  The domain
    method :py:meth:`~.Domain.unify` is used to find a domain that can
    represent all the elements of two given domains.

    >>> from sympy import ZZ, QQ, symbols
    >>> x, y = symbols('x, y')
    >>> ZZ.unify(QQ)
    QQ
    >>> ZZ[x].unify(QQ)
    QQ[x]
    >>> ZZ[x].unify(QQ[y])
    QQ[x,y]

    If a domain is a :py:class:`~.Ring` then is might have an associated
    :py:class:`~.Field` and vice versa. The :py:meth:`~.Domain.get_field` and
    :py:meth:`~.Domain.get_ring` methods will find or create the associated
    domain.

    >>> from sympy import ZZ, QQ, Symbol
```

```
    >>> x = Symbol('x')
    >>> ZZ.has_assoc_Field
    True
    >>> ZZ.get_field()
    QQ
    >>> QQ.has_assoc_Ring
    True
    >>> QQ.get_ring()
    ZZ
    >>> K = QQ[x]
    >>> K
    QQ[x]
    >>> K.get_field()
    QQ(x)

    See also
    ========

    DomainElement: abstract base class for domain elements
    construct_domain: construct a minimal domain for some expressions

    """

    dtype = None         # type: Optional[Type]
    """The type (class) of the elements of this :py:class:`~.Domain`:

    >>> from sympy import ZZ, QQ, Symbol
    >>> ZZ.dtype
    <class 'int'>
    >>> z = ZZ(2)
    >>> z
    2
    >>> type(z)
    <class 'int'>
    >>> type(z) == ZZ.dtype
    True

    Every domain has an associated **dtype** ("datatype")
```

```
    which is the
    class of the associated domain elements.

    See also
    ========

    of_type
    """

    zero = None          # type: Optional[Any]
    """The zero element of the :py:class:`~.Domain`:

    >>> from sympy import QQ
    >>> QQ.zero
    0
    >>> QQ.of_type(QQ.zero)
    True

    See also
    ========

    of_type
    one
    """

    one = None           # type: Optional[Any]
    """The one element of the :py:class:`~.Domain`:

    >>> from sympy import QQ
    >>> QQ.one
    1
    >>> QQ.of_type(QQ.one)
    True

    See also
    ========

    of_type
    zero
    """

```

```
    is_Ring = False
    """Boolean flag indicating if the domain is
    a :py:class:`~.Ring`.

    >>> from sympy import ZZ
    >>> ZZ.is_Ring
    True

    Basically every :py:class:`~.Domain` represents a
    ring so this flag is
    not that useful.

    See also
    ========

    is_PID
    is_Field
    get_ring
    has_assoc_Ring
    """

    is_Field = False
    """Boolean flag indicating if the domain is
    a :py:class:`~.Field`.

    >>> from sympy import ZZ, QQ
    >>> ZZ.is_Field
    False
    >>> QQ.is_Field
    True

    See also
    ========

    is_PID
    is_Ring
    get_field
    has_assoc_Field
    """

    has_assoc_Ring = False
```

```
    """Boolean flag indicating if the domain has an associated
    :py:class:`~.Ring`.

    >>> from sympy import QQ
    >>> QQ.has_assoc_Ring
    True
    >>> QQ.get_ring()
    ZZ

    See also
    ========

    is_Field
    get_ring
    """

    has_assoc_Field = False
    """Boolean flag indicating if the domain has an associated
    :py:class:`~.Field`.

    >>> from sympy import ZZ
    >>> ZZ.has_assoc_Field
    True
    >>> ZZ.get_field()
    QQ

    See also
    ========

    is_Field
    get_field
    """

    is_FiniteField = is_FF = False
    is_IntegerRing = is_ZZ = False
    is_RationalField = is_QQ = False
    is_GaussianRing = is_ZZ_I = False
    is_GaussianField = is_QQ_I = False
    is_RealField = is_RR = False
```

```
    is_ComplexField = is_CC = False
    is_AlgebraicField = is_Algebraic = False
    is_PolynomialRing = is_Poly = False
    is_FractionField = is_Frac = False
    is_SymbolicDomain = is_EX = False
    is_SymbolicRawDomain = is_EXRAW = False
    is_FiniteExtension = False

    is_Exact = True
    is_Numerical = False

    is_Simple = False
    is_Composite = False

    is_PID = False
    """Boolean flag indicating if the domain is a
    `principal ideal domain`_.

    >>> from sympy import ZZ
    >>> ZZ.has_assoc_Field
    True
    >>> ZZ.get_field()
    QQ

    .. _principal ideal domain: https://en.wikipedia.org/
    wiki/Principal_ideal_domain

    See also
    ========

    is_Field
    get_field
    """

    has_CharacteristicZero = False

    rep = None  # type: Optional[str]
    alias = None  # type: Optional[str]

    def __init__(self):
        raise NotImplementedError
```

```python

    def __str__(self):
        return self.rep

    def __repr__(self):
        return str(self)

    def __hash__(self):
        return hash((self.__class__.__name__, self.dtype))

    def new(self, *args):
        return self.dtype(*args)

    @property
    def tp(self):
        """Alias for :py:attr:`~.Domain.dtype`"""
        return self.dtype

    def __call__(self, *args):
        """Construct an element of ``self`` domain from ``args``. """
        return self.new(*args)

    def normal(self, *args):
        return self.dtype(*args)

    def convert_from(self, element, base):
        """Convert ``element`` to ``self.dtype`` given the base domain. """
        if base.alias is not None:
            method = "from_" + base.alias
        else:
            method = "from_" + base.__class__.__name__

        _convert = getattr(self, method)

        if _convert is not None:
            result = _convert(element, base)

            if result is not None:
```

```
                return result

        raise CoercionFailed("Cannot convert %s of type %s from %s to %s" % (element, type(element), base, self))

    def convert(self, element, base=None):
        """Convert ``element`` to ``self.dtype``. """

        if base is not None:
            if _not_a_coeff(element):
                raise CoercionFailed('%s is not in any domain' % element)
            return self.convert_from(element, base)

        if self.of_type(element):
            return element

        if _not_a_coeff(element):
            raise CoercionFailed('%s is not in any domain' % element)

        from sympy.polys.domains import ZZ, QQ, RealField, ComplexField

        if ZZ.of_type(element):
            return self.convert_from(element, ZZ)

        if isinstance(element, int):
            return self.convert_from(ZZ(element), ZZ)

        if HAS_GMPY:
            integers = ZZ
            if isinstance(element, integers.tp):
                return self.convert_from(element, integers)

            rationals = QQ
            if isinstance(element, rationals.tp):
                return self.convert_from(element, rationals)
```

```python

        if isinstance(element, float):
            parent = RealField(tol=False)
            return self.convert_from(parent(element), parent)

        if isinstance(element, complex):
            parent = ComplexField(tol=False)
            return self.convert_from(parent(element), parent)

        if isinstance(element, DomainElement):
            return self.convert_from(element, element.parent())

        # TODO: implement this in from_ methods
        if self.is_Numerical and getattr(element, 'is_ground', False):
            return self.convert(element.LC())

        if isinstance(element, Basic):
            try:
                return self.from_sympy(element)
            except (TypeError, ValueError):
                pass
        else: # TODO: remove this branch
            if not is_sequence(element):
                try:
                    element = sympify(element, strict=True)
                    if isinstance(element, Basic):
                        return self.from_sympy(element)
                except (TypeError, ValueError):
                    pass

        raise CoercionFailed("Cannot convert %s of type %s to %s" % (element, type(element), self))

    def of_type(self, element):
        """Check if ``a`` is of type ``dtype``. """
        return isinstance(element, self.tp) # XXX: this
```

```
        isn't correct, e.g. PolyElement

    def __contains__(self, a):
        """Check if ``a`` belongs to this domain. """
        try:
            if _not_a_coeff(a):
                raise CoercionFailed
            self.convert(a)  # this might raise, too
        except CoercionFailed:
            return False

        return True

    def to_sympy(self, a):
        """Convert domain element *a* to a SymPy expression (Expr).

        Explanation
        ===========

        Convert a :py:class:`~.Domain` element *a* to :py:class:`~.Expr`. Most
        public SymPy functions work with objects of type :py:class:`~.Expr`.
        The elements of a :py:class:`~.Domain` have a different internal
        representation. It is not possible to mix domain elements with
        :py:class:`~.Expr` so each domain has :py:meth:`~.Domain.to_sympy` and
        :py:meth:`~.Domain.from_sympy` methods to convert its domain elements
        to and from :py:class:`~.Expr`.

        Parameters
        ==========

        a: domain element
            An element of this :py:class:`~.Domain`.

        Returns
```

```
        =======

        expr: Expr
            A normal SymPy expression of
            type :py:class:`~.Expr`.

        Examples
        ========

        Construct an element of the :ref:`QQ` domain and
        then convert it to
        :py:class:`~.Expr`.

        >>> from sympy import QQ, Expr
        >>> q_domain = QQ(2)
        >>> q_domain
        2
        >>> q_expr = QQ.to_sympy(q_domain)
        >>> q_expr
        2

        Although the printed forms look similar these
        objects are not of the
        same type.

        >>> isinstance(q_domain, Expr)
        False
        >>> isinstance(q_expr, Expr)
        True

        Construct an element of :ref:`K[x]` and convert
        to
        :py:class:`~.Expr`.

        >>> from sympy import Symbol
        >>> x = Symbol('x')
        >>> K = QQ[x]
        >>> x_domain = K.gens[0]  # generator x as a
        domain element
        >>> p_domain = x_domain**2/3 + 1
        >>> p_domain
```

```
        1/3*x**2 + 1
        >>> p_expr = K.to_sympy(p_domain)
        >>> p_expr
        x**2/3 + 1

        The :py:meth:`~.Domain.from_sympy` method is used for the opposite
        conversion from a normal SymPy expression to a domain element.

        >>> p_domain == p_expr
        False
        >>> K.from_sympy(p_expr) == p_domain
        True
        >>> K.to_sympy(p_domain) == p_expr
        True
        >>> K.from_sympy(K.to_sympy(p_domain)) == p_domain
        True
        >>> K.to_sympy(K.from_sympy(p_expr)) == p_expr
        True

        The :py:meth:`~.Domain.from_sympy` method makes it easier to construct
        domain elements interactively.

        >>> from sympy import Symbol
        >>> x = Symbol('x')
        >>> K = QQ[x]
        >>> K.from_sympy(x**2/3 + 1)
        1/3*x**2 + 1

        See also
        ========

        from_sympy
        convert_from
        """
        raise NotImplementedError

    def from_sympy(self, a):
```

```
        """Convert a SymPy expression to an element of
        this domain.

        Explanation
        ===========

        See :py:meth:`~.Domain.to_sympy` for explanation
        and examples.

        Parameters
        ==========

        expr: Expr
            A normal SymPy expression of
            type :py:class:`~.Expr`.

        Returns
        =======

        a: domain element
            An element of this :py:class:`~.Domain`.

        See also
        ========

        to_sympy
        convert_from
        """
        raise NotImplementedError

    def sum(self, args):
        return sum(args)

    def from_FF(K1, a, K0):
        """Convert ``ModularInteger(int)`` to ``dtype``.
        """
        return None

    def from_FF_python(K1, a, K0):
        """Convert ``ModularInteger(int)`` to ``dtype``.
        """
```

```
            return None

        def from_ZZ_python(K1, a, K0):
            """Convert a Python ``int`` object to ``dtype``. """
            return None

        def from_QQ_python(K1, a, K0):
            """Convert a Python ``Fraction`` object to ``dtype``. """
            return None

        def from_FF_gmpy(K1, a, K0):
            """Convert ``ModularInteger(mpz)`` to ``dtype``. """
            return None

        def from_ZZ_gmpy(K1, a, K0):
            """Convert a GMPY ``mpz`` object to ``dtype``. """
            return None

        def from_QQ_gmpy(K1, a, K0):
            """Convert a GMPY ``mpq`` object to ``dtype``. """
            return None

        def from_RealField(K1, a, K0):
            """Convert a real element object to ``dtype``. """
            return None

        def from_ComplexField(K1, a, K0):
            """Convert a complex element to ``dtype``. """
            return None

        def from_AlgebraicField(K1, a, K0):
            """Convert an algebraic number to ``dtype``. """
            return None

        def from_PolynomialRing(K1, a, K0):
```

```
        """Convert a polynomial to ``dtype``. """
        if a.is_ground:
            return K1.convert(a.LC, K0.dom)

    def from_FractionField(K1, a, K0):
        """Convert a rational function to ``dtype``. """
        return None

    def from_MonogenicFiniteExtension(K1, a, K0):
        """Convert an ``ExtensionElement`` to ``dtype``."""
        return K1.convert_from(a.rep, K0.ring)

    def from_ExpressionDomain(K1, a, K0):
        """Convert a ``EX`` object to ``dtype``. """
        return K1.from_sympy(a.ex)

    def from_ExpressionRawDomain(K1, a, K0):
        """Convert a ``EX`` object to ``dtype``. """
        return K1.from_sympy(a)

    def from_GlobalPolynomialRing(K1, a, K0):
        """Convert a polynomial to ``dtype``. """
        if a.degree() <= 0:
            return K1.convert(a.LC(), K0.dom)

    def from_GeneralizedPolynomialRing(K1, a, K0):
        return K1.from_FractionField(a, K0)

    def unify_with_symbols(K0, K1, symbols):
        if (K0.is_Composite and (set(K0.symbols) & set(symbols))) or (K1.is_Composite and (set(K1.symbols) & set(symbols))):
            raise UnificationFailed("Cannot unify %s with %s, given %s generators" % (K0, K1, tuple(symbols)))

        return K0.unify(K1)

    def unify(K0, K1, symbols=None):
        """
```

```
        Construct a minimal domain that contains elements of ``K0`` and ``K1``.

        Known domains (from smallest to largest):

        - ``GF(p)``
        - ``ZZ``
        - ``QQ``
        - ``RR(prec, tol)``
        - ``CC(prec, tol)``
        - ``ALG(a, b, c)``
        - ``K[x, y, z]``
        - ``K(x, y, z)``
        - ``EX``

        """
        if symbols is not None:
            return K0.unify_with_symbols(K1, symbols)

        if K0 == K1:
            return K0

        if K0.is_EXRAW:
            return K0
        if K1.is_EXRAW:
            return K1

        if K0.is_EX:
            return K0
        if K1.is_EX:
            return K1

        if K0.is_FiniteExtension or K1.is_FiniteExtension:
            if K1.is_FiniteExtension:
                K0, K1 = K1, K0
            if K1.is_FiniteExtension:
                # Unifying two extensions.
                # Try to ensure that K0.unify(K1) == K1.unify(K0)
                if list(ordered([K0.modulus,
```

```
                K1.modulus]))[1] == K0.modulus:
                    K0, K1 = K1, K0
                return K1.set_domain(K0)
            else:
                # Drop the generator from other and unify with the base domain
                K1 = K1.drop(K0.symbol)
                K1 = K0.domain.unify(K1)
                return K0.set_domain(K1)

        if K0.is_Composite or K1.is_Composite:
            K0_ground = K0.dom if K0.is_Composite else K0
            K1_ground = K1.dom if K1.is_Composite else K1

            K0_symbols = K0.symbols if K0.is_Composite else ()
            K1_symbols = K1.symbols if K1.is_Composite else ()

            domain = K0_ground.unify(K1_ground)
            symbols = _unify_gens(K0_symbols, K1_symbols)
            order = K0.order if K0.is_Composite else K1.order

            if ((K0.is_FractionField and K1.is_PolynomialRing or
                 K1.is_FractionField and K0.is_PolynomialRing) and
                 (not K0_ground.is_Field or not K1_ground.is_Field) and domain.is_Field
                 and domain.has_assoc_Ring):
                domain = domain.get_ring()

            if K0.is_Composite and (not K1.is_Composite or K0.is_FractionField or K1.is_PolynomialRing):
                cls = K0.__class__
            else:
                cls = K1.__class__

            from sympy.polys.domains.old_polynomialring
```

```
            import GlobalPolynomialRing
            if cls == GlobalPolynomialRing:
                return cls(domain, symbols)

            return cls(domain, symbols, order)

        def mkinexact(cls, K0, K1):
            prec = max(K0.precision, K1.precision)
            tol = max(K0.tolerance, K1.tolerance)
            return cls(prec=prec, tol=tol)

        if K1.is_ComplexField:
            K0, K1 = K1, K0
        if K0.is_ComplexField:
            if K1.is_ComplexField or K1.is_RealField:
                return mkinexact(K0.__class__, K0, K1)
            else:
                return K0

        if K1.is_RealField:
            K0, K1 = K1, K0
        if K0.is_RealField:
            if K1.is_RealField:
                return mkinexact(K0.__class__, K0, K1)
            elif K1.is_GaussianRing or K1.is_GaussianField:
                from sympy.polys.domains.complexfield import ComplexField
                return ComplexField(prec=K0.precision, tol=K0.tolerance)
            else:
                return K0

        if K1.is_AlgebraicField:
            K0, K1 = K1, K0
        if K0.is_AlgebraicField:
            if K1.is_GaussianRing:
                K1 = K1.get_field()
            if K1.is_GaussianField:
                K1 = K1.as_AlgebraicField()
            if K1.is_AlgebraicField:
```

```python
                return K0.__class__(K0.dom.unify(K1.dom),
                *_unify_gens(K0.orig_ext, K1.orig_ext))
            else:
                return K0

        if K0.is_GaussianField:
            return K0
        if K1.is_GaussianField:
            return K1

        if K0.is_GaussianRing:
            if K1.is_RationalField:
                K0 = K0.get_field()
            return K0
        if K1.is_GaussianRing:
            if K0.is_RationalField:
                K1 = K1.get_field()
            return K1

        if K0.is_RationalField:
            return K0
        if K1.is_RationalField:
            return K1

        if K0.is_IntegerRing:
            return K0
        if K1.is_IntegerRing:
            return K1

        if K0.is_FiniteField and K1.is_FiniteField:
            return K0.__class__(max(K0.mod, K1.mod,
            key=default_sort_key))

        from sympy.polys.domains import EX
        return EX

    def __eq__(self, other):
        """Returns ``True`` if two domains are
        equivalent. """
        return isinstance(other, Domain) and self.dtype
        == other.dtype
```

```python

    def __ne__(self, other):
        """Returns ``False`` if two domains are equivalent. """
        return not self == other

    def map(self, seq):
        """Rersively apply ``self`` to all elements of ``seq``. """
        result = []

        for elt in seq:
            if isinstance(elt, list):
                result.append(self.map(elt))
            else:
                result.append(self(elt))

        return result

    def get_ring(self):
        """Returns a ring associated with ``self``. """
        raise DomainError('there is no ring associated with %s' % self)

    def get_field(self):
        """Returns a field associated with ``self``. """
        raise DomainError('there is no field associated with %s' % self)

    def get_exact(self):
        """Returns an exact domain associated with ``self``. """
        return self

    def __getitem__(self, symbols):
        """The mathematical way to make a polynomial ring. """
        if hasattr(symbols, '__iter__'):
            return self.poly_ring(*symbols)
        else:
            return self.poly_ring(symbols)
```

```python

    def poly_ring(self, *symbols, order=lex):
        """Returns a polynomial ring, i.e. `K[X]`. """
        from sympy.polys.domains.polynomialring import PolynomialRing
        return PolynomialRing(self, symbols, order)

    def frac_field(self, *symbols, order=lex):
        """Returns a fraction field, i.e. `K(X)`. """
        from sympy.polys.domains.fractionfield import FractionField
        return FractionField(self, symbols, order)

    def old_poly_ring(self, *symbols, **kwargs):
        """Returns a polynomial ring, i.e. `K[X]`. """
        from sympy.polys.domains.old_polynomialring import PolynomialRing
        return PolynomialRing(self, *symbols, **kwargs)

    def old_frac_field(self, *symbols, **kwargs):
        """Returns a fraction field, i.e. `K(X)`. """
        from sympy.polys.domains.old_fractionfield import FractionField
        return FractionField(self, *symbols, **kwargs)

    def algebraic_field(self, *extension, alias=None):
        r"""Returns an algebraic field, i.e. `K(\alpha, \ldots)`. """
        raise DomainError("Cannot create algebraic field over %s" % self)

    def alg_field_from_poly(self, poly, alias=None, root_index=-1):
        r"""
        Convenience method to construct an algebraic extension on a root of a
        polynomial, chosen by root index.

        Parameters
        ==========

```

```
        poly : :py:class:`~.Poly`
            The polynomial whose root generates the
            extension.
        alias : str, optional (default=None)
            Symbol name for the generator of the
            extension.
            E.g. "alpha" or "theta".
        root_index : int, optional (default=-1)
            Specifies which root of the polynomial is
            desired. The ordering is
            as defined by the :py:class:`~.ComplexRootOf`
            class. The default of
            ``-1`` selects the most natural choice in the
            common cases of
            quadratic and cyclotomic fields (the square
            root on the positive
            real or imaginary axis, resp. $\mathrm{e}
            ^{2\pi i/n}$).

        Examples
        ========

        >>> from sympy import QQ, Poly
        >>> from sympy.abc import x
        >>> f = Poly(x**2 - 2)
        >>> K = QQ.alg_field_from_poly(f)
        >>> K.ext.minpoly == f
        True
        >>> g = Poly(8*x**3 - 6*x - 1)
        >>> L = QQ.alg_field_from_poly(g, "alpha")
        >>> L.ext.minpoly == g
        True
        >>> L.to_sympy(L([1, 1, 1]))
        alpha**2 + alpha + 1

        """
        from sympy.polys.rootoftools import CRootOf
        root = CRootOf(poly, root_index)
        alpha = AlgebraicNumber(root, alias=alias)
        return self.algebraic_field(alpha, alias=alias)

```

```
    def cyclotomic_field(self, n, ss=False, alias="zeta", gen=None, root_index=-1):
        r"""
        Convenience method to construct a cyclotomic field.

        Parameters
        ==========

        n : int
            Construct the nth cyclotomic field.
        ss : boolean, optional (default=False)
            If True, append *n* as a subscript on the alias string.
        alias : str, optional (default="zeta")
            Symbol name for the generator.
        gen : :py:class:`~.Symbol`, optional (default=None)
            Desired variable for the cyclotomic polynomial that defines the
            field. If ``None``, a dummy variable will be used.
        root_index : int, optional (default=-1)
            Specifies which root of the polynomial is desired. The ordering is
            as defined by the :py:class:`~.ComplexRootOf` class. The default of
            ``-1`` selects the root $\mathrm{e}^{2\pi i/n}$.

        Examples
        ========

        >>> from sympy import QQ, latex
        >>> K = QQ.cyclotomic_field(5)
        >>> K.to_sympy(K([-1, 1]))
        1 - zeta
        >>> L = QQ.cyclotomic_field(7, True)
        >>> a = L.to_sympy(L([-1, 1]))
        >>> print(a)
        1 - zeta7
```

```
        >>> print(latex(a))
        1 - \zeta_{7}

        """
        from sympy.polys.specialpolys import cyclotomic_poly
        if ss:
            alias += str(n)
        return self.alg_field_from_poly(cyclotomic_poly(n, gen), alias=alias,
                                        root_index=root_index)

    def inject(self, *symbols):
        """Inject generators into this domain. """
        raise NotImplementedError

    def drop(self, *symbols):
        """Drop generators from this domain. """
        if self.is_Simple:
            return self
        #raise NotImplementedError  # pragma: no cover

    def is_zero(self, a):
        """Returns True if ``a`` is zero. """
        return not a

    def is_one(self, a):
        """Returns True if ``a`` is one. """
        return a == self.one

    def is_positive(self, a):
        """Returns True if ``a`` is positive. """
        return a > 0

    def is_negative(self, a):
        """Returns True if ``a`` is negative. """
        return a < 0

    def is_nonpositive(self, a):
```

```
        """Returns True if ``a`` is non-positive. """
        return a <= 0

    def is_nonnegative(self, a):
        """Returns True if ``a`` is non-negative. """
        return a >= 0

    def canonical_unit(self, a):
        if self.is_negative(a):
            return -self.one
        else:
            return self.one

    def abs(self, a):
        """Absolute value of ``a``, implies ``__abs__``.
        """
        return abs(a)

    def neg(self, a):
        """Returns ``a`` negated, implies ``__neg__``.
        """
        return -a

    def pos(self, a):
        """Returns ``a`` positive, implies ``__pos__``.
        """
        return +a

    def add(self, a, b):
        """Sum of ``a`` and ``b``, implies ``__add__``.
        """
        return a + b

    def sub(self, a, b):
        """Difference of ``a`` and ``b``, implies
        ``__sub__``.  """
        return a - b

    def mul(self, a, b):
        """Product of ``a`` and ``b``, implies
        ``__mul__``.  """
```

```
            return a * b

        def pow(self, a, b):
            """Raise ``a`` to power ``b``, implies
            ``__pow__``.  """
            return a ** b

        def exquo(self, a, b):
            """Exact quotient of *a* and *b*. Analogue of ``a
            / b``.

            Explanation
            ===========

            This is essentially the same as ``a / b`` except
            that an error will be
            raised if the division is inexact (if there is
            any remainder) and the
            result will always be a domain element. When
            working in a
            :py:class:`~.Domain` that is not
            a :py:class:`~.Field` (e.g. :ref:`ZZ`
            or :ref:`K[x]`) ``exquo`` should be used instead
            of ``/``.

            The key invariant is that if ``q = K.exquo(a,
            b)`` (and ``exquo`` does
            not raise an exception) then ``a == b*q``.

            Examples
            ========

            We can use ``K.exquo`` instead of ``/`` for exact
            division.

            >>> from sympy import ZZ
            >>> ZZ.exquo(ZZ(4), ZZ(2))
            2
            >>> ZZ.exquo(ZZ(5), ZZ(2))
            Traceback (most recent call last):
                ...
```

```
        ExactQuotientFailed: 2 does not divide 5 in ZZ

        Over a :py:class:`~.Field` such as :ref:`QQ`, division (with nonzero
        divisor) is always exact so in that case ``/`` can be used instead of
        :py:meth:`~.Domain.exquo`.

        >>> from sympy import QQ
        >>> QQ.exquo(QQ(5), QQ(2))
        5/2
        >>> QQ(5) / QQ(2)
        5/2

        Parameters
        ==========

        a: domain element
            The dividend
        b: domain element
            The divisor

        Returns
        =======

        q: domain element
            The exact quotient

        Raises
        ======

        ExactQuotientFailed: if exact division is not possible.
        ZeroDivisionError: when the divisor is zero.

        See also
        ========

        quo: Analogue of ``a // b``
        rem: Analogue of ``a % b``
        div: Analogue of ``divmod(a, b)``
```

```

        Notes
        =====

        Since the default :py:attr:`~.Domain.dtype` for :ref:`ZZ` is ``int``
        (or ``mpz``) division as ``a / b`` should not be used as it would give
        a ``float``.

        >>> ZZ(4) / ZZ(2)
        2.0
        >>> ZZ(5) / ZZ(2)
        2.5

        Using ``/`` with :ref:`ZZ` will lead to incorrect results so
        :py:meth:`~.Domain.exquo` should be used instead.

        """
        raise NotImplementedError

    def quo(self, a, b):
        """Quotient of *a* and *b*. Analogue of ``a // b``.

        ``K.quo(a, b)`` is equivalent to ``K.div(a, b)[0]``. See
        :py:meth:`~.Domain.div` for more explanation.

        See also
        ========

        rem: Analogue of ``a % b``
        div: Analogue of ``divmod(a, b)``
        exquo: Analogue of ``a / b``
        """
        raise NotImplementedError

    def rem(self, a, b):
        """Modulo division of *a* and *b*. Analogue of
```

```
        ``a % b``.

        ``K.rem(a, b)`` is equivalent to ``K.div(a, b)
        [1]``. See
        :py:meth:`~.Domain.div` for more explanation.

        See also
        ========

        quo: Analogue of ``a // b``
        div: Analogue of ``divmod(a, b)``
        exquo: Analogue of ``a / b``
        """
        raise NotImplementedError

    def div(self, a, b):
        """Quotient and remainder for *a* and *b*.
        Analogue of ``divmod(a, b)``

        Explanation
        ===========

        This is essentially the same as ``divmod(a, b)``
        except that is more
        consistent when working over
        some :py:class:`~.Field` domains such as
        :ref:`QQ`. When working over an
        arbitrary :py:class:`~.Domain` the
        :py:meth:`~.Domain.div` method should be used
        instead of ``divmod``.

        The key invariant is that if ``q, r = K.div(a,
        b)`` then
        ``a == b*q + r``.

        The result of ``K.div(a, b)`` is the same as the
        tuple
        ``(K.quo(a, b), K.rem(a, b))`` except that if
        both quotient and
        remainder are needed then it is more efficient to
        use
```

```
        :py:meth:`~.Domain.div`.

        Examples
        ========

        We can use ``K.div`` instead of ``divmod`` for floor division and
        remainder.

        >>> from sympy import ZZ, QQ
        >>> ZZ.div(ZZ(5), ZZ(2))
        (2, 1)

        If ``K`` is a :py:class:`~.Field` then the division is always exact
        with a remainder of :py:attr:`~.Domain.zero`.

        >>> QQ.div(QQ(5), QQ(2))
        (5/2, 0)

        Parameters
        ==========

        a: domain element
            The dividend
        b: domain element
            The divisor

        Returns
        =======

        (q, r): tuple of domain elements
            The quotient and remainder

        Raises
        ======

        ZeroDivisionError: when the divisor is zero.

        See also
        ========
```

```

        quo: Analogue of ``a // b``
        rem: Analogue of ``a % b``
        exquo: Analogue of ``a / b``

        Notes
        =====

        If ``gmpy`` is installed then the ``gmpy.mpq`` type will be used as
        the :py:attr:`~.Domain.dtype` for :ref:`QQ`. The ``gmpy.mpq`` type
        defines ``divmod`` in a way that is undesirable so
        :py:meth:`~.Domain.div` should be used instead of ``divmod``.

        >>> a = QQ(1)
        >>> b = QQ(3, 2)
        >>> a               # doctest: +SKIP
        mpq(1,1)
        >>> b               # doctest: +SKIP
        mpq(3,2)
        >>> divmod(a, b)    # doctest: +SKIP
        (mpz(0), mpq(1,1))
        >>> QQ.div(a, b)    # doctest: +SKIP
        (mpq(2,3), mpq(0,1))

        Using ``//`` or ``%`` with :ref:`QQ` will lead to incorrect results so
        :py:meth:`~.Domain.div` should be used instead.

        """
        raise NotImplementedError

    def invert(self, a, b):
        """Returns inversion of ``a mod b``, implies something. """
        raise NotImplementedError

    def revert(self, a):
```

```
        """Returns ``a**(-1)`` if possible. """
        raise NotImplementedError

    def numer(self, a):
        """Returns numerator of ``a``. """
        raise NotImplementedError

    def denom(self, a):
        """Returns denominator of ``a``. """
        raise NotImplementedError

    def half_gcdex(self, a, b):
        """Half extended GCD of ``a`` and ``b``. """
        s, t, h = self.gcdex(a, b)
        return s, h

    def gcdex(self, a, b):
        """Extended GCD of ``a`` and ``b``. """
        raise NotImplementedError

    def cofactors(self, a, b):
        """Returns GCD and cofactors of ``a`` and ``b``.
        """
        gcd = self.gcd(a, b)
        cfa = self.quo(a, gcd)
        cfb = self.quo(b, gcd)
        return gcd, cfa, cfb

    def gcd(self, a, b):
        """Returns GCD of ``a`` and ``b``. """
        raise NotImplementedError

    def lcm(self, a, b):
        """Returns LCM of ``a`` and ``b``. """
        raise NotImplementedError

    def log(self, a, b):
        """Returns b-base logarithm of ``a``. """
        raise NotImplementedError

    def sqrt(self, a):
```

```
        """Returns square root of ``a``. """
        raise NotImplementedError

    def evalf(self, a, prec=None, **options):
        """Returns numerical approximation of ``a``. """
        return self.to_sympy(a).evalf(prec, **options)

    n = evalf

    def real(self, a):
        return a

    def imag(self, a):
        return self.zero

    def almosteq(self, a, b, tolerance=None):
        """Check if ``a`` and ``b`` are almost equal. """
        return a == b

    def characteristic(self):
        """Return the characteristic of this domain. """
        raise NotImplementedError('characteristic()')


__all__ = ['Domain']
```