

```

1  #LyX
2  """Implementation of :class:`Domain` class. """
3
4
5  from typing import Any, Optional, Type
6
7  from sympy.core.numbers import AlgebraicNumber
8  from sympy.core import Basic, sympify
9  from sympy.core.sorting import default_sort_key, ordered
10 from sympy.external.gmpy import HAS_GMPY
11 from sympy.polys.domains.domainelement import
DomainElement
12 from sympy.polys.orderings import lex
13 from sympy.polys.polyerrors import UnificationFailed,
CoercionFailed, DomainError
14 from sympy.polys.polyutils import _unify_gens,
_not_a_coeff
15 from sympy.utilities import public
16 from sympy.utilities.iterables import is_sequence
17
18
19 @public
20 class Domain:
21     """Superclass for all domains in the polys domains
system.
22
23     See :ref:`polys-domainsintro` for an introductory
explanation of the
24     domains system.
25
26     The :py:class:`~.Domain` class is an abstract base
class for all of the
27     concrete domain types. There are many
different :py:class:`~.Domain`
28     subclasses each of which has an associated ``dtype``
which is a class
29     representing the elements of the domain. The
coefficients of a
30     :py:class:`~.Poly` are elements of a domain which
must be a subclass of
31     :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
103 class (``dtype``) for the elements of the domain. For
    example the
104 :ref:`K[x]` domains are represented by a domain
    object which is an
105 instance of :py:class:`~.PolynomialRing` and the
    elements are always
106 instances of :py:class:`~.PolyElement`. The
    implementation class
107 represents particular types of mathematical
    expressions in a way that is
108 more efficient than a normal SymPy expression which
    is of type
109 :py:class:`~.Expr`. The domain
    methods :py:meth:`~.Domain.from_sympy` and
110 :py:meth:`~.Domain.to_sympy` are used to convert from
    :py:class:`~.Expr`
111 to a domain element and vice versa.
112
113 >>> from sympy import Symbol, ZZ, Expr
114 >>> x = Symbol('x')
115 >>> K = ZZ[x]                # polynomial ring domain
116 >>> K
117 ZZ[x]
118 >>> type(K)                  # class of the domain
119 <class
    'sympy.polys.domains.polynomialring.PolynomialRing'>
120 >>> K.dtype                  # class of the elements
121 <class 'sympy.polys.rings.PolyElement'>
122 >>> p_expr = x**2 + 1      # Expr
123 >>> p_expr
124 x**2 + 1
125 >>> type(p_expr)
126 <class 'sympy.core.add.Add'>
127 >>> isinstance(p_expr, Expr)
128 True
129 >>> p_domain = K.from_sympy(p_expr)
130 >>> p_domain                  # domain element
131 x**2 + 1
132 >>> type(p_domain)
133 <class 'sympy.polys.rings.PolyElement'>
```

```
134 >>> K.to_sympy(p_domain) == p_expr
135 True
136 The :py:meth:`~.Domain.convert_from` method is used
    to convert domain
137 elements from one domain to another.
138
139 >>> from sympy import ZZ, QQ
140 >>> ez = ZZ(2)
141 >>> eq = QQ.convert_from(ez, ZZ)
142 >>> type(ez) # doctest: +SKIP
143 <class 'int'>
144 >>> type(eq) # doctest: +SKIP
145 <class
    'sympy.polys.domains.pythonrational.PythonRational'>
146
147 Elements from different domains should not be mixed
    in arithmetic or other
148 operations: they should be converted to a common
    domain first. The domain
149 method :py:meth:`~.Domain.unify` is used to find a
    domain that can
150 represent all the elements of two given domains.
151
152 >>> from sympy import ZZ, QQ, symbols
153 >>> x, y = symbols('x, y')
154 >>> ZZ.unify(QQ)
155 QQ
156 >>> ZZ[x].unify(QQ)
157 QQ[x]
158 >>> ZZ[x].unify(QQ[y])
159 QQ[x,y]
160
161 If a domain is a :py:class:`~.Ring` then it might
    have an associated
162 :py:class:`~.Field` and vice versa.
    The :py:meth:`~.Domain.get_field` and
163 :py:meth:`~.Domain.get_ring` methods will find or
    create the associated
164 domain.
165
166 >>> from sympy import ZZ, QQ, Symbol
```

```

167     >>> x = Symbol('x')
168     >>> ZZ.has_assoc_Field
169     True
170     >>> ZZ.get_field()
171     QQ
172     >>> QQ.has_assoc_Ring
173     True
174     >>> QQ.get_ring()
175     ZZ
176     >>> K = QQ[x]
177     >>> K
178     QQ[x]
179     >>> K.get_field()
180     QQ(x)
181
182     See also
183     =====
184
185     DomainElement: abstract base class for domain
186     construct_domain: construct a minimal domain for some
187     expressions
188
189     """
190     dtype = None # type: Optional[Type]
191     """The type (class) of the elements of
192     this :py:class:`~.Domain`:
193
194     >>> from sympy import ZZ, QQ, Symbol
195     >>> ZZ.dtype
196     <class 'int'>
197     >>> z = ZZ(2)
198     >>> z
199     2
200     >>> type(z)
201     <class 'int'>
202     >>> type(z) == ZZ.dtype
203     True
204
205     Every domain has an associated **dtype** ("datatype")

```

```
205     which is the
206     class of the associated domain elements.
207
208     See also
209     =====
210
211     of_type
212     """
213
214     zero = None # type: Optional[Any]
215     """The zero element of the :py:class:`~.Domain`:
```

```
216
217     >>> from sympy import QQ
218     >>> QQ.zero
219     0
220     >>> QQ.of_type(QQ.zero)
221     True
222
223     See also
224     =====
225
226     of_type
227     one
228     """
229
230     one = None # type: Optional[Any]
231     """The one element of the :py:class:`~.Domain`:
```

```
232
233     >>> from sympy import QQ
234     >>> QQ.one
235     1
236     >>> QQ.of_type(QQ.one)
237     True
238
239     See also
240     =====
241
242     of_type
243     zero
244     """
```

```
245     is_Ring = False
246     """Boolean flag indicating if the domain is
247     a :py:class:`~.Ring`.
248
249     >>> from sympy import ZZ
250     >>> ZZ.is_Ring
251     True
252
253     Basically every :py:class:`~.Domain` represents a
254     ring so this flag is
255     not that useful.
256
257     See also
258     =====
259
260     is_PID
261     is_Field
262     get_ring
263     has_assoc_Ring
264     """
265
266     is_Field = False
267     """Boolean flag indicating if the domain is
268     a :py:class:`~.Field`.
269
270     >>> from sympy import ZZ, QQ
271     >>> ZZ.is_Field
272     False
273     >>> QQ.is_Field
274     True
275
276     See also
277     =====
278
279     is_PID
280     is_Ring
281     get_field
282     has_assoc_Field
283     """
284
285     has_assoc_Ring = False
```



```
283     """Boolean flag indicating if the domain has an
284     associated
285     :py:class:`~.Ring`.
286
287     >>> from sympy import QQ
288     >>> QQ.has_assoc_Ring
289     True
290     >>> QQ.get_ring()
291     ZZ
292
293     See also
294     =====
295
296     is_Field
297     get_ring
298     """
299
300     has_assoc_Field = False
301     """Boolean flag indicating if the domain has an
302     associated
303     :py:class:`~.Field`.
304
305     >>> from sympy import ZZ
306     >>> ZZ.has_assoc_Field
307     True
308     >>> ZZ.get_field()
309     QQ
310
311     See also
312     =====
313
314     is_Field
315     get_field
316     """
317
318     is_FiniteField = is_FF = False
319     is_IntegerRing = is_ZZ = False
320     is_RationalField = is_QQ = False
321     is_GaussianRing = is_ZZ_I = False
322     is_GaussianField = is_QQ_I = False
323     is_RealField = is_RR = False
```

```

322     is_ComplexField = is_CC = False
323     is_AlgebraicField = is_Algebraic = False
324     is_PolynomialRing = is_Poly = False
325     is_FractionField = is_Frac = False
326     is_SymbolicDomain = is_EX = False
327     is_SymbolicRawDomain = is_EXRAW = False
328     is_FiniteExtension = False
329
330     is_Exact = True
331     is_Numerical = False
332
333     is_Simple = False
334     is_Composite = False
335
336     is_PID = False
337     """Boolean flag indicating if the domain is a
    `principal ideal domain`_.
338
339     >>> from sympy import ZZ
340     >>> ZZ.has_assoc_Field
341     True
342     >>> ZZ.get_field()
343     QQ
344
345     .. _principal ideal domain: https://en.wikipedia.org/
wiki/Principal_ideal_domain
346
347     See also
348     =====
349
350     is_Field
351     get_field
352     """
353
354     has_CharacteristicZero = False
355
356     rep = None # type: Optional[str]
357     alias = None # type: Optional[str]
358
359     def __init__(self):
360         raise NotImplementedError

```

```
361
362     def __str__(self):
363         return self.rep
364
365     def __repr__(self):
366         return str(self)
367
368     def __hash__(self):
369         return hash((self.__class__.__name__,
370                     self.dtype))
371
372     def new(self, *args):
373         return self.dtype(*args)
374
375     @property
376     def tp(self):
377         """Alias for :py:attr:`~.Domain.dtype`"""
378         return self.dtype
379
380     def __call__(self, *args):
381         """Construct an element of ``self`` domain from
382         ``args``. """
383         return self.new(*args)
384
385     def normal(self, *args):
386         return self.dtype(*args)
387
388     def convert_from(self, element, base):
389         """Convert ``element`` to ``self.dtype`` given
390         the base domain. """
391         if base.alias is not None:
392             method = "from_" + base.alias
393         else:
394             method = "from_" + base.__class__.__name__
395
396         _convert = getattr(self, method)
397
398         if _convert is not None:
399             result = _convert(element, base)
400
401             if result is not None:
```

```
399         return result
400
401     raise CoercionFailed("Cannot convert %s of type
%s from %s to %s" % (element, type(element),
base, self))
402
403     def convert(self, element, base=None):
404         """Convert ``element`` to ``self.dtype``. """
405
406         if base is not None:
407             if _not_a_coeff(element):
408                 raise CoercionFailed('%s is not in any
domain' % element)
409             return self.convert_from(element, base)
410
411         if self.of_type(element):
412             return element
413
414         if _not_a_coeff(element):
415             raise CoercionFailed('%s is not in any
domain' % element)
416
417         from sympy.polys.domains import ZZ, QQ,
RealField, ComplexField
418
419         if ZZ.of_type(element):
420             return self.convert_from(element, ZZ)
421
422         if isinstance(element, int):
423             return self.convert_from(ZZ(element), ZZ)
424
425         if HAS_GMPY:
426             integers = ZZ
427             if isinstance(element, integers.tp):
428                 return self.convert_from(element,
integers)
429
430             rationals = QQ
431             if isinstance(element, rationals.tp):
432                 return self.convert_from(element,
rationals)
```

```
433
434     if isinstance(element, float):
435         parent = RealField(tol=False)
436         return self.convert_from(parent(element),
437                                   parent)
438
439     if isinstance(element, complex):
440         parent = ComplexField(tol=False)
441         return self.convert_from(parent(element),
442                                   parent)
443
444     if isinstance(element, DomainElement):
445         return self.convert_from(element,
446                                   element.parent())
447
448     # TODO: implement this in from_ methods
449     if self.is_Numerical and getattr(element,
450 'is_ground', False):
451         return self.convert(element.LC())
452
453     if isinstance(element, Basic):
454         try:
455             return self.from_sympy(element)
456         except (TypeError, ValueError):
457             pass
458     else: # TODO: remove this branch
459         if not is_sequence(element):
460             try:
461                 element = sympify(element,
462 strict=True)
463                 if isinstance(element, Basic):
464                     return self.from_sympy(element)
465             except (TypeError, ValueError):
466                 pass
467
468     raise CoercionFailed("Cannot convert %s of type
469 %s to %s" % (element, type(element), self))
470
471 def of_type(self, element):
472     """Check if ``a`` is of type ``dtype``. """
473     return isinstance(element, self.tp) # XXX: this
```

isn't correct, e.g. PolyElement

```

468
469 def __contains__(self, a):
470     """Check if ``a`` belongs to this domain. """
471     try:
472         if _not_a_coeff(a):
473             raise CoercionFailed
474         self.convert(a) # this might raise, too
475     except CoercionFailed:
476         return False
477
478     return True
479
480 def to_sympy(self, a):
481     """Convert domain element *a* to a SymPy
482     expression (Expr).
483
484     Explanation
485     =====
486
487     Convert a :py:class:`~.Domain` element *a*
488     to :py:class:`~.Expr`. Most
489     public SymPy functions work with objects of
490     type :py:class:`~.Expr`.
491     The elements of a :py:class:`~.Domain` have a
492     different internal
493     representation. It is not possible to mix domain
494     elements with
495     :py:class:`~.Expr` so each domain
496     has :py:meth:`~.Domain.to_sympy` and
497     :py:meth:`~.Domain.from_sympy` methods to convert
498     its domain elements
499     to and from :py:class:`~.Expr`.
500
501     Parameters
502     =====
503
504     a: domain element
505         An element of this :py:class:`~.Domain`.
506
507     Returns

```

```
501         =====
502
503         expr: Expr
504         A normal SymPy expression of
505         type :py:class:`~.Expr`.
506
507         Examples
508         =====
509
510         Construct an element of the :ref:`QQ` domain and
511         then convert it to
512         :py:class:`~.Expr`.
513
514         >>> from sympy import QQ, Expr
515         >>> q_domain = QQ(2)
516         >>> q_domain
517         2
518         >>> q_expr = QQ.to_sympy(q_domain)
519         >>> q_expr
520         2
521
522         Although the printed forms look similar these
523         objects are not of the
524         same type.
525
526         >>> isinstance(q_domain, Expr)
527         False
528         >>> isinstance(q_expr, Expr)
529         True
530
531         Construct an element of :ref:`K[x]` and convert
532         to
533         :py:class:`~.Expr`.
534
535         >>> from sympy import Symbol
536         >>> x = Symbol('x')
```

```

537         1/3*x**2 + 1
538     >>> p_expr = K.to_sympy(p_domain)
539     >>> p_expr
540     x**2/3 + 1
541
542     The :py:meth:`~.Domain.from_sympy` method is used
543     for the opposite
544     conversion from a normal SymPy expression to a
545     domain element.
546
547     >>> p_domain == p_expr
548     False
549     >>> K.from_sympy(p_expr) == p_domain
550     True
551     >>> K.to_sympy(p_domain) == p_expr
552     True
553     >>> K.from_sympy(K.to_sympy(p_domain)) ==
554     p_domain
555     True
556     >>> K.to_sympy(K.from_sympy(p_expr)) == p_expr
557     True
558
559     The :py:meth:`~.Domain.from_sympy` method makes
560     it easier to construct
561     domain elements interactively.
562
563     >>> from sympy import Symbol
564     >>> x = Symbol('x')
565     >>> K = QQ[x]
566     >>> K.from_sympy(x**2/3 + 1)
567     1/3*x**2 + 1
568
569     See also
570     =====
571
572     from_sympy
573     convert_from
574     """
575     raise NotImplementedError
576
577 def from_sympy(self, a):

```



```

574         """Convert a SymPy expression to an element of
575         this domain.
576
577         Explanation
578         =====
579
580         See :py:meth:`~.Domain.to_sympy` for explanation
581         and examples.
582
583         Parameters
584         =====
585
586         expr: Expr
587             A normal SymPy expression of
588             type :py:class:`~.Expr`.
589
590         Returns
591         =====
592
593         a: domain element
594             An element of this :py:class:`~.Domain`.
595
596         See also
597         =====
598
599         to_sympy
600         convert_from
601         """
602         raise NotImplementedError
603
604     def sum(self, args):
605         return sum(args)
606
607     def from_FF(K1, a, K0):
608         """Convert ``ModularInteger(int)`` to ``dtype``.
609         """
610         return None
611
612     def from_FF_python(K1, a, K0):
613         """Convert ``ModularInteger(int)`` to ``dtype``.
614         """

```

```
610         return None
611
612     def from_ZZ_python(K1, a, K0):
613         """Convert a Python ``int`` object to ``dtype``.
614         """
615         return None
616
617     def from_QQ_python(K1, a, K0):
618         """Convert a Python ``Fraction`` object to
619         ``dtype``. """
620         return None
621
622     def from_FF_gmpy(K1, a, K0):
623         """Convert ``ModularInteger(mpz)`` to ``dtype``.
624         """
625         return None
626
627     def from_ZZ_gmpy(K1, a, K0):
628         """Convert a GMPY ``mpz`` object to ``dtype``.
629         """
630         return None
631
632     def from_QQ_gmpy(K1, a, K0):
633         """Convert a GMPY ``mpq`` object to ``dtype``.
634         """
635         return None
636
637     def from_RealField(K1, a, K0):
638         """Convert a real element object to ``dtype``.
639         """
640         return None
641
642     def from_ComplexField(K1, a, K0):
643         """Convert a complex element to ``dtype``. """
644         return None
645
646     def from_AlgebraicField(K1, a, K0):
647         """Convert an algebraic number to ``dtype``. """
648         return None
649
650     def from_PolynomialRing(K1, a, K0):
```

```

645         """Convert a polynomial to ``dtype``. """
646         if a.is_ground:
647             return K1.convert(a.LC, K0.dom)
648
649     def from_FractionField(K1, a, K0):
650         """Convert a rational function to ``dtype``. """
651         return None
652
653     def from_MonogenicFiniteExtension(K1, a, K0):
654         """Convert an ``ExtensionElement`` to ``dtype``. """
655         return K1.convert_from(a.rep, K0.ring)
656
657     def from_ExpressionDomain(K1, a, K0):
658         """Convert a ``EX`` object to ``dtype``. """
659         return K1.from_sympy(a.ex)
660
661     def from_ExpressionRawDomain(K1, a, K0):
662         """Convert a ``EX`` object to ``dtype``. """
663         return K1.from_sympy(a)
664
665     def from_GlobalPolynomialRing(K1, a, K0):
666         """Convert a polynomial to ``dtype``. """
667         if a.degree() <= 0:
668             return K1.convert(a.LC(), K0.dom)
669
670     def from_GeneralizedPolynomialRing(K1, a, K0):
671         return K1.from_FractionField(a, K0)
672
673     def unify_with_symbols(K0, K1, symbols):
674         if (K0.is_Composite and (set(K0.symbols) &
675                                 set(symbols))) or (K1.is_Composite and
676                                                     (set(K1.symbols) & set(symbols))):
677             raise UnificationFailed("Cannot unify %s with
678                                     %s, given %s generators" % (K0, K1,
679                                                                     tuple(symbols)))
680
681         return K0.unify(K1)
682
683     def unify(K0, K1, symbols=None):
684         """

```

```

681         Construct a minimal domain that contains elements
682         of ``K0`` and ``K1``.
683
684         Known domains (from smallest to largest):
685
686         - ``GF(p)``
687         - ``ZZ``
688         - ``QQ``
689         - ``RR(prec, tol)``
690         - ``CC(prec, tol)``
691         - ``ALG(a, b, c)``
692         - ``K[x, y, z]``
693         - ``K(x, y, z)``
694         - ``EX``
695
696         """
697         if symbols is not None:
698             return K0.unify_with_symbols(K1, symbols)
699
700         if K0 == K1:
701             return K0
702
703         if K0.is_EXRAW:
704             return K0
705         if K1.is_EXRAW:
706             return K1
707
708         if K0.is_EX:
709             return K0
710         if K1.is_EX:
711             return K1
712
713         if K0.is_FiniteExtension or
714         K1.is_FiniteExtension:
715             if K1.is_FiniteExtension:
716                 K0, K1 = K1, K0
717             if K1.is_FiniteExtension:
718                 # Unifying two extensions.
719                 # Try to ensure that K0.unify(K1) ==
720                 K1.unify(K0)
721                 if list(ordered([K0.modulus,

```

```

719         K1.modulus]))[1] == K0.modulus:
720             K0, K1 = K1, K0
721             return K1.set_domain(K0)
722         else:
723             # Drop the generator from other and unify
724             with the base domain
725             K1 = K1.drop(K0.symbol)
726             K1 = K0.domain.unify(K1)
727             return K0.set_domain(K1)
728
729     if K0.is_Composite or K1.is_Composite:
730         K0_ground = K0.dom if K0.is_Composite else K0
731         K1_ground = K1.dom if K1.is_Composite else K1
732
733         K0_symbols = K0.symbols if K0.is_Composite
734         else ()
735         K1_symbols = K1.symbols if K1.is_Composite
736         else ()
737
738         domain = K0_ground.unify(K1_ground)
739         symbols = _unify_gens(K0_symbols, K1_symbols)
740         order = K0.order if K0.is_Composite else
741         K1.order
742
743         if ((K0.is_FractionField and
744             K1.is_PolynomialRing or
745             K1.is_FractionField and
746             K0.is_PolynomialRing) and
747             (not K0_ground.is_Field or not
748             K1_ground.is_Field) and domain.is_Field
749             and domain.has_assoc_Ring):
750             domain = domain.get_ring()
751
752         if K0.is_Composite and (not K1.is_Composite
753             or K0.is_FractionField or
754             K1.is_PolynomialRing):
755             cls = K0.__class__
756         else:
757             cls = K1.__class__
758
759         from sympy.polys.domains.old_polynomialring

```

```
import GlobalPolynomialRing
750 if cls == GlobalPolynomialRing:
751     return cls(domain, symbols)
752
753     return cls(domain, symbols, order)
754
755 def mkinexact(cls, K0, K1):
756     prec = max(K0.precision, K1.precision)
757     tol = max(K0.tolerance, K1.tolerance)
758     return cls(prec=prec, tol=tol)
759
760 if K1.is_ComplexField:
761     K0, K1 = K1, K0
762 if K0.is_ComplexField:
763     if K1.is_ComplexField or K1.is_RealField:
764         return mkinexact(K0.__class__, K0, K1)
765     else:
766         return K0
767
768 if K1.is_RealField:
769     K0, K1 = K1, K0
770 if K0.is_RealField:
771     if K1.is_RealField:
772         return mkinexact(K0.__class__, K0, K1)
773     elif K1.is_GaussianRing or
774           K1.is_GaussianField:
775         from sympy.polys.domains.complexfield
776         import ComplexField
777         return ComplexField(prec=K0.precision,
778                             tol=K0.tolerance)
779     else:
780         return K0
781
782 if K1.is_AlgebraicField:
783     K0, K1 = K1, K0
784 if K0.is_AlgebraicField:
785     if K1.is_GaussianRing:
786         K1 = K1.get_field()
787     if K1.is_GaussianField:
788         K1 = K1.as_AlgebraicField()
789     if K1.is_AlgebraicField:
```

```
787         return K0.__class__(K0.dom.unify(K1.dom),
788                               *_unify_gens(K0.orig_ext, K1.orig_ext))
789     else:
790         return K0
791
792     if K0.is_GaussianField:
793         return K0
794     if K1.is_GaussianField:
795         return K1
796
797     if K0.is_GaussianRing:
798         if K1.is_RationalField:
799             K0 = K0.get_field()
800             return K0
801     if K1.is_GaussianRing:
802         if K0.is_RationalField:
803             K1 = K1.get_field()
804             return K1
805
806     if K0.is_RationalField:
807         return K0
808     if K1.is_RationalField:
809         return K1
810
811     if K0.is_IntegerRing:
812         return K0
813     if K1.is_IntegerRing:
814         return K1
815
816     if K0.is_FiniteField and K1.is_FiniteField:
817         return K0.__class__(max(K0.mod, K1.mod,
818                                   key=default_sort_key))
819
820     from sympy.polys.domains import EX
821     return EX
822
823 def __eq__(self, other):
824     """Returns ``True`` if two domains are
825     equivalent. """
826     return isinstance(other, Domain) and self.dtype
827     == other.dtype
```

```
824
825     def __ne__(self, other):
826         """Returns ``False`` if two domains are
            equivalent. """
827         return not self == other
828
829     def map(self, seq):
830         """Rersively apply ``self`` to all elements of
            ``seq``. """
831         result = []
832
833         for elt in seq:
834             if isinstance(elt, list):
835                 result.append(self.map(elt))
836             else:
837                 result.append(self(elt))
838
839         return result
840
841     def get_ring(self):
842         """Returns a ring associated with ``self``. """
843         raise DomainError('there is no ring associated
            with %s' % self)
844
845     def get_field(self):
846         """Returns a field associated with ``self``. """
847         raise DomainError('there is no field associated
            with %s' % self)
848
849     def get_exact(self):
850         """Returns an exact domain associated with
            ``self``. """
851         return self
852
853     def __getitem__(self, symbols):
854         """The mathematical way to make a polynomial
            ring. """
855         if hasattr(symbols, '__iter__'):
856             return self.poly_ring(*symbols)
857         else:
858             return self.poly_ring(symbols)
```



```

859
860 def poly_ring(self, *symbols, order=lex):
861     """Returns a polynomial ring, i.e. `K[X]`. """
862     from sympy.polys.domains.polynomialring import
      PolynomialRing
863     return PolynomialRing(self, symbols, order)
864
865 def frac_field(self, *symbols, order=lex):
866     """Returns a fraction field, i.e. `K(X)`. """
867     from sympy.polys.domains.fractionfield import
      FractionField
868     return FractionField(self, symbols, order)
869
870 def old_poly_ring(self, *symbols, **kwargs):
871     """Returns a polynomial ring, i.e. `K[X]`. """
872     from sympy.polys.domains.old_polynomialring
      import PolynomialRing
873     return PolynomialRing(self, *symbols, **kwargs)
874
875 def old_frac_field(self, *symbols, **kwargs):
876     """Returns a fraction field, i.e. `K(X)`. """
877     from sympy.polys.domains.old_fractionfield import
      FractionField
878     return FractionField(self, *symbols, **kwargs)
879
880 def algebraic_field(self, *extension, alias=None):
881     r"""Returns an algebraic field, i.e. `K(\alpha,
      \ldots)`. """
882     raise DomainError("Cannot create algebraic field
      over %s" % self)
883
884 def alg_field_from_poly(self, poly, alias=None,
      root_index=-1):
885     r"""
886     Convenience method to construct an algebraic
887     extension on a root of a
888     polynomial, chosen by root index.
889
890     Parameters
891     =====

```

```

892     poly : :py:class:`~.Poly`
893         The polynomial whose root generates the
           extension.
894     alias : str, optional (default=None)
895         Symbol name for the generator of the
           extension.
896         E.g. "alpha" or "theta".
897     root_index : int, optional (default=-1)
898         Specifies which root of the polynomial is
           desired. The ordering is
899         as defined by the :py:class:`~.ComplexRootOf`
           class. The default of
900         ``-1`` selects the most natural choice in the
           common cases of
901         quadratic and cyclotomic fields (the square
           root on the positive
902         real or imaginary axis, resp.  $e^{\pm 2\pi i/n}$ ).
903
904     Examples
905     =====
906
907     >>> from sympy import QQ, Poly
908     >>> from sympy.abc import x
909     >>> f = Poly(x**2 - 2)
910     >>> K = QQ.alg_field_from_poly(f)
911     >>> K.ext.minpoly == f
912     True
913     >>> g = Poly(8*x**3 - 6*x - 1)
914     >>> L = QQ.alg_field_from_poly(g, "alpha")
915     >>> L.ext.minpoly == g
916     True
917     >>> L.to_sympy(L([1, 1, 1]))
918     alpha**2 + alpha + 1
919
920     """"
921     from sympy.polys.rootoftools import CRootOf
922     root = CRootOf(poly, root_index)
923     alpha = AlgebraicNumber(root, alias=alias)
924     return self.algebraic_field(alpha, alias=alias)
925

```

```

926     def cyclotomic_field(self, n, ss=False, alias="zeta",
927                          gen=None, root_index=-1):
928         r"""
929         Convenience method to construct a cyclotomic
930         field.
931
932         Parameters
933         =====
934
935         n : int
936         Construct the nth cyclotomic field.
937         ss : boolean, optional (default=False)
938         If True, append *n* as a subscript on the
939         alias string.
940         alias : str, optional (default="zeta")
941         Symbol name for the generator.
942         gen : :py:class:`~.Symbol`, optional
943         (default=None)
944         Desired variable for the cyclotomic
945         polynomial that defines the
946         field. If ``None``, a dummy variable will be
947         used.
948         root_index : int, optional (default=-1)
949         Specifies which root of the polynomial is
950         desired. The ordering is
951         as defined by the :py:class:`~.ComplexRootOf`
952         class. The default of
953         ``-1`` selects the root  $e^{2\pi i/n}$ .
954
955         Examples
956         =====
957
958         >>> from sympy import QQ, latex
959         >>> K = QQ.cyclotomic_field(5)
960         >>> K.to_sympy(K([-1, 1]))
961         1 - zeta
962         >>> L = QQ.cyclotomic_field(7, True)
963         >>> a = L.to_sympy(L([-1, 1]))
964         >>> print(a)
965         1 - zeta7

```

```

958     >>> print(latex(a))
959     1 - \zeta_{7}
960
961     """
962     from sympy.polys.specialpolys import
cycloatomic_poly
963     if ss:
964         alias += str(n)
965     return
self.alg_field_from_poly(cycloatomic_poly(n, gen),
alias=alias,
966
                                root_index=root_index)
967
def inject(self, *symbols):
968     """Inject generators into this domain. """
969     raise NotImplementedError
970
971
def drop(self, *symbols):
972     """Drop generators from this domain. """
973     if self.is_Simple:
974         return self
975     #raise NotImplementedError # pragma: no cover
976
977
def is_zero(self, a):
978     """Returns True if ``a`` is zero. """
979     return not a
980
981
def is_one(self, a):
982     """Returns True if ``a`` is one. """
983     return a == self.one
984
985
def is_positive(self, a):
986     """Returns True if ``a`` is positive. """
987     return a > 0
988
989
def is_negative(self, a):
990     """Returns True if ``a`` is negative. """
991     return a < 0
992
993
def is_nonpositive(self, a):
994

```

```
995         """Returns True if ``a`` is non-positive. """
996         return a <= 0
997
998     def is_nonnegative(self, a):
999         """Returns True if ``a`` is non-negative. """
1000         return a >= 0
1001
1002     def canonical_unit(self, a):
1003         if self.is_negative(a):
1004             return -self.one
1005         else:
1006             return self.one
1007
1008     def abs(self, a):
1009         """Absolute value of ``a``, implies ``__abs__``.
1010         """
1011         return abs(a)
1012
1013     def neg(self, a):
1014         """Returns ``a`` negated, implies ``__neg__``.
1015         """
1016         return -a
1017
1018     def pos(self, a):
1019         """Returns ``a`` positive, implies ``__pos__``.
1020         """
1021         return +a
1022
1023     def add(self, a, b):
1024         """Sum of ``a`` and ``b``, implies ``__add__``.
1025         """
1026         return a + b
1027
1028     def sub(self, a, b):
1029         """Difference of ``a`` and ``b``, implies
1030         ``__sub__``. """
1031         return a - b
1032
1033     def mul(self, a, b):
1034         """Product of ``a`` and ``b``, implies
1035         ``__mul__``. """
```

```

1030         return a * b
1031
1032     def pow(self, a, b):
1033         """Raise ``a`` to power ``b``, implies
1034             ``__pow__``. """
1035         return a ** b
1036
1037     def exquo(self, a, b):
1038         """Exact quotient of *a* and *b*. Analogue of ``a
1039             / b``.
1040
1041             Explanation
1042             =====
1043
1044             This is essentially the same as ``a / b`` except
1045             that an error will be
1046             raised if the division is inexact (if there is
1047             any remainder) and the
1048             result will always be a domain element. When
1049             working in a
1050             :py:class:`~.Domain` that is not
1051             a :py:class:`~.Field` (e.g. :ref:`ZZ`
1052             or :ref:`K[x]`) ``exquo`` should be used instead
1053             of ``/``.
1054
1055             The key invariant is that if ``q = K.exquo(a,
1056             b)`` (and ``exquo`` does
1057             not raise an exception) then ``a == b*q``.
1058
1059             Examples
1060             =====
1061
1062             We can use ``K.exquo`` instead of ``/`` for exact
1063             division.
1064
1065             >>> from sympy import ZZ
1066             >>> ZZ.exquo(ZZ(4), ZZ(2))
1067             2
1068             >>> ZZ.exquo(ZZ(5), ZZ(2))
1069             Traceback (most recent call last):
1070             ...

```

```

1062     ExactQuotientFailed: 2 does not divide 5 in ZZ
1063
1064     Over a :py:class:`~.Field` such as :ref:`QQ`,
1065     division (with nonzero
1066     divisor) is always exact so in that case ``/``
1067     can be used instead of
1068     :py:meth:`~.Domain.exquo`.
1069
1070     >>> from sympy import QQ
1071     >>> QQ.exquo(QQ(5), QQ(2))
1072     5/2
1073
1074     Parameters
1075     =====
1076
1077     a: domain element
1078         The dividend
1079     b: domain element
1080         The divisor
1081
1082     Returns
1083     =====
1084
1085     q: domain element
1086         The exact quotient
1087
1088     Raises
1089     =====
1090
1091     ExactQuotientFailed: if exact division is not
1092     possible.
1093     ZeroDivisionError: when the divisor is zero.
1094
1095     See also
1096     =====
1097
1098     quo: Analogue of ``a // b``
1099     rem: Analogue of ``a % b``
1100     div: Analogue of ``divmod(a, b)``

```

```

1100
1101     Notes
1102     =====
1103
1104     Since the default :py:attr:`~.Domain.dtype`
1105     for :ref:`ZZ` is ``int``
1106     (or ``mpz``) division as ``a / b`` should not be
1107     used as it would give
1108     a ``float``.
1109
1110     >>> ZZ(4) / ZZ(2)
1111     2.0
1112     >>> ZZ(5) / ZZ(2)
1113     2.5
1114
1115     Using ``/`` with :ref:`ZZ` will lead to incorrect
1116     results so
1117     :py:meth:`~.Domain.exquo` should be used instead.
1118
1119     """
1120     raise NotImplementedError
1121
1122     def quo(self, a, b):
1123         """Quotient of *a* and *b*. Analogue of ``a //
1124         b``.
1125
1126         ``K.quo(a, b)`` is equivalent to ``K.div(a, b)
1127         [0]``. See
1128         :py:meth:`~.Domain.div` for more explanation.
1129
1130         See also
1131         =====
1132
1133         rem: Analogue of ``a % b``
1134         div: Analogue of ``divmod(a, b)``
1135         exquo: Analogue of ``a / b``
1136         """
1137         raise NotImplementedError
1138
1139     def rem(self, a, b):
1140         """Modulo division of *a* and *b*. Analogue of

```



```

1136         ``a % b``.
1137         ``K.rem(a, b)`` is equivalent to ``K.div(a, b)
1138         [1]``. See
1139         :py:meth:`~.Domain.div` for more explanation.
1140
1141         See also
1142         =====
1143
1144         quo: Analogue of ``a // b``
1145         div: Analogue of ``divmod(a, b)``
1146         exquo: Analogue of ``a / b``
1147         """
1148         raise NotImplementedError
1149
1150     def div(self, a, b):
1151         """Quotient and remainder for *a* and *b*.
1152         Analogue of ``divmod(a, b)``
1153
1154         Explanation
1155         =====
1156
1157         This is essentially the same as ``divmod(a, b)``
1158         except that is more
1159         consistent when working over
1160         some :py:class:`~.Field` domains such as
1161         :ref:`QQ`. When working over an
1162         arbitrary :py:class:`~.Domain` the
1163         :py:meth:`~.Domain.div` method should be used
1164         instead of ``divmod``.
1165
1166         The key invariant is that if ``q, r = K.div(a,
1167         b)`` then
1168         ``a == b*q + r``.
1169
1170         The result of ``K.div(a, b)`` is the same as the
1171         tuple
1172         ``(K.quo(a, b), K.rem(a, b))`` except that if
1173         both quotient and
1174         remainder are needed then it is more efficient to
1175         use

```

```
1166         :py:meth:`~.Domain.div`.
1167
1168     Examples
1169     =====
1170
1171     We can use ``K.div`` instead of ``divmod`` for
1172     floor division and
1173     remainder.
1174
1175     >>> from sympy import ZZ, QQ
1176     >>> ZZ.div(ZZ(5), ZZ(2))
1177     (2, 1)
1178
1179     If ``K`` is a :py:class:`~.Field` then the
1180     division is always exact
1181     with a remainder of :py:attr:`~.Domain.zero`.
1182
1183     >>> QQ.div(QQ(5), QQ(2))
1184     (5/2, 0)
1185
1186     Parameters
1187     =====
1188
1189     a: domain element
1190         The dividend
1191     b: domain element
1192         The divisor
1193
1194     Returns
1195     =====
1196
1197     (q, r): tuple of domain elements
1198         The quotient and remainder
1199
1200     Raises
1201     =====
1202
1203     ZeroDivisionError: when the divisor is zero.
1204
1205     See also
1206     =====
```

```

1205
1206     quo: Analogue of ``a // b``
1207     rem: Analogue of ``a % b``
1208     exquo: Analogue of ``a / b``
1209
1210     Notes
1211     =====
1212
1213     If ``gmpy`` is installed then the ``gmpy.mpq``
1214     type will be used as
1215     the :py:attr:`~.Domain.dtype` for :ref:`QQ`. The
1216     ``gmpy.mpq`` type
1217     defines ``divmod`` in a way that is undesirable
1218     so
1219     :py:meth:`~.Domain.div` should be used instead of
1220     ``divmod``.
1221
1222     >>> a = QQ(1)
1223     >>> b = QQ(3, 2)
1224     >>> a                                # doctest: +SKIP
1225     mpq(1,1)
1226     >>> b                                # doctest: +SKIP
1227     mpq(3,2)
1228     >>> divmod(a, b)                      # doctest: +SKIP
1229     (mpz(0), mpq(1,1))
1230     >>> QQ.div(a, b)                      # doctest: +SKIP
1231     (mpq(2,3), mpq(0,1))
1232
1233     Using ``//`` or ``%`` with :ref:`QQ` will lead to
1234     incorrect results so
1235     :py:meth:`~.Domain.div` should be used instead.
1236
1237     """
1238     raise NotImplementedError
1239
1240     def invert(self, a, b):
1241         """Returns inversion of ``a mod b``, implies
1242         something. """
1243         raise NotImplementedError
1244
1245     def revert(self, a):

```

```
1240         """Returns ``a*(-1)`` if possible. """
1241         raise NotImplementedError
1242
1243     def numer(self, a):
1244         """Returns numerator of ``a``. """
1245         raise NotImplementedError
1246
1247     def denom(self, a):
1248         """Returns denominator of ``a``. """
1249         raise NotImplementedError
1250
1251     def half_gcdex(self, a, b):
1252         """Half extended GCD of ``a`` and ``b``. """
1253         s, t, h = self.gcdex(a, b)
1254         return s, h
1255
1256     def gcdex(self, a, b):
1257         """Extended GCD of ``a`` and ``b``. """
1258         raise NotImplementedError
1259
1260     def cofactors(self, a, b):
1261         """Returns GCD and cofactors of ``a`` and ``b``.
1262         """
1263         gcd = self.gcd(a, b)
1264         cfa = self.quo(a, gcd)
1265         cfb = self.quo(b, gcd)
1266         return gcd, cfa, cfb
1267
1268     def gcd(self, a, b):
1269         """Returns GCD of ``a`` and ``b``. """
1270         raise NotImplementedError
1271
1272     def lcm(self, a, b):
1273         """Returns LCM of ``a`` and ``b``. """
1274         raise NotImplementedError
1275
1276     def log(self, a, b):
1277         """Returns b-base logarithm of ``a``. """
1278         raise NotImplementedError
1279
1280     def sqrt(self, a):
```

```
1280         """Returns square root of ``a``. """
1281         raise NotImplementedError
1282
1283     def evalf(self, a, prec=None, **options):
1284         """Returns numerical approximation of ``a``. """
1285         return self.to_sympy(a).evalf(prec, **options)
1286
1287     n = evalf
1288
1289     def real(self, a):
1290         return a
1291
1292     def imag(self, a):
1293         return self.zero
1294
1295     def almosteq(self, a, b, tolerance=None):
1296         """Check if ``a`` and ``b`` are almost equal. """
1297         return a == b
1298
1299     def characteristic(self):
1300         """Return the characteristic of this domain. """
1301         raise NotImplementedError('characteristic()')
1302
1303
1304 __all__ = ['Domain']
1305
```