5 Computational Domains

Writing mathematics on paper or on the blackboard requires a compromise between ease of notations and rigour. The same holds for the day-to-day use of a computer algebra system. Sage tries to give this choice to the user, by letting her/him specify, more or less rigorously, the computational domains: what is the nature of the considered objects, in which sets do they live, which operations can be applied to them?

5.1 Sage is Object-Oriented

Python and Sage use heavily the object-oriented programming paradigm. Even though this remains relatively transparent in common use, it is useful to know a little about this paradigm, which is quite natural in a mathematical context.

5.1.1 Objects, Classes and Methods

The object-oriented programming paradigm consists in modelling each physical or abstract entity one wishes to manipulate by a programming language construction called an *object*. In most cases, as in Python, each object is an instance of a *class*. For example, the rational number 12/35 is represented by an object which is an instance of the `Rational` class:

```
sage: o = 12/35
sage: type(o)
<type 'sage.rings.rational.Rational'>
```

Note that this class is really associated to the object 12/35, and not to the variable `o` in which it is stored:

```
sage: type(12/35)
<type 'sage.rings.rational.Rational'>
```

Let us be more precise. An *object* is a part of the computer memory which stores the required information to represent the corresponding entity. The *class* in turn defines two things:

1. the *data structure* of an object, i.e., how the information is organised in memory. For example, the `Rational` class specifies that a rational number like 12/35 is represented by two integers: its numerator and its denominator;

2. its *behaviour*, in particular the available *operations* on this object: how to obtain the numerator of a rational number, how to compute its absolute value, how to multiply or add two rational numbers. Each of these operations is implemented by a *method* (here respectively `numer`, `abs`, `__mul__`, `__add__`).

To factor an integer, we will thus call the `factor` method with the following syntax:

```
sage: o = 720
sage: o.factor()
2^4 * 3^2 * 5
```

which we can read as follows: "take the value of `o` and apply to it the `factor` method, without any other argument". Under the hood, Python performs the following computation:

```
sage: type(o).factor(o)
2^4 * 3^2 * 5
```

From left to right: "request from the class of `o` (`type(o)`) the factorisation method (`type(o).factor`), and apply it to `o`".

Please note that we can apply this method not only to a variable, but also directly to a value:

```
sage: 720.factor()
2^4 * 3^2 * 5
```

and thus we can chain the operations, from left to right. Here, we first take the numerator of a rational number, then we factor this numerator:

```
sage: o = 720 / 133
sage: o.numerator().factor()
2^4 * 3^2 * 5
```

To make the user's life easier, Sage also provides a *function* `factor`, so that `factor(o)` is a shortcut for `o.factor()`. It is the case for several common functions, and it is possible to add our own shortcuts, as illustrated in the following exercise.

Exercise 18. Build a shortcut `ndigits` so that `ndigits(o)` calls the `ndigits` method of the object `o`.

5.1.2 Objects and Polymorphism

Almost all Sage operations are *polymorphic*, i.e., they apply to several kinds of objects. For example, whatever the nature of the object `o` that we want to "factor", we will use the same notation `o.factor()` (or its shortcut `factor(o)`). The computations to be performed however differ to factor an integer or a polynomial! They also differ if the polynomial has rational coefficients, or coefficients in a finite field. The object class determines the version of the `factor` code that will be called.

Similarly, like the usual mathematical notation, the product of two objects `a` and `b` can always be denoted `a*b`, even if the algorithm used differs in each case[1]. Here is a product of two integers:

```
sage: 3 * 7
21
```

a product of two rational numbers, obtained by multiplying the numerators and denominators, then reducing the fraction:

```
sage: (2/3) * (6/5)
4/5
```

a product of two complex numbers, using the relation $i^2 = -1$:

```
sage: (1 + I)  *  (1 - I)
2
```

some commutative products of two formal expressions:

```
sage: (x + 2) * (x + 1)
(x + 2)*(x + 1)
sage: (x + 1) * (x + 2)
(x + 2)*(x + 1)
```

Apart from the notation simplicity, this form of polymorphism enables us to write *generic* programs which apply to any object having the involved operations (here multiplication):

```
sage: def fourth_power(a):
....:     a = a * a
....:     a = a * a
....:     return a
```

```
sage: fourth_power(2)
16
sage: fourth_power(3/2)
81/16
```

[1]For a binary operation like the product, the selection of the appropriate method is slightly more complex than what was described above. Indeed, we might deal with mixed operations like the sum $2 + 3/4$ of an integer and of a rational number. In this case, 2 will be converted in the rational $2/1$, and the addition of two rationals will be called. The rules that describe which operand must be converted, and how it should be converted, are part of the *coercion model*.

```
sage: fourth_power(I)
1
sage: fourth_power(x+1)
(x + 1)^4
sage: M = matrix([[0,-1],[1,0]]); M
[ 0 -1]
[ 1  0]
sage: fourth_power(M)
[1 0]
[0 1]
```

5.1.3 Introspection

Python objects, and therefore Sage objects, have some *introspection* features. This means that, during execution, we can "ask" an object for its class, its methods, etc., and manipulate the obtained informations using the usual constructions of the programming language. For instance, the class of an object `o` is itself a Python object, and we can obtain it using `type(o)`:

```
sage: t = type(5/1); t
<type 'sage.rings.rational.Rational'>
sage: t == type(5)
False
```

We see here that the expression `5/1` constructs the rational number 5, which differs — as Python object — from the integer 5!

The introspection tools also give access to the factorisation on-line help from an object of integer type:

```
sage: o = 720
sage: o.factor?
Docstring:
     Return the prime factorization of this integer as a formal
     Factorization object.
...
```

and even to the source code of that function:

```
sage: o.factor??
...
def factor(self, algorithm='pari', proof=None, ...)
      ...
      if algorithm == 'pari':
          ...
      elif algorithm in ['kash', 'magma']:
          ...
```

Avoiding some technical details, we see here that Sage delegates the integer factorisation to other tools (PARI/GP, Kash, or Magma).

In the same vein, we can use automatic completion to interactively "ask" an object `o` which operations can be applied to it:

```
sage: o.n<tab>
o.n                    o.nbits                o.ndigits
o.next_prime           o.next_prime_power     o.next_probable_prime
o.nth_root             o.numerator            o.numerical_approx
```

Once again, it is a form of introspection.

5.2 Elements, Parents, Categories

5.2.1 Elements and Parents

In the preceding section, we have seen the concept of *class* of an object. In practice, it is enough to know that this notion exists; we rarely have to explicitly look for the type of an object. However, Sage introduces another concept closer to mathematics: the *parent* of an object, that we will detail now.

Assume for example that we want to know if an element a is *invertible*. The answer does not only depend on the element itself, but also on the mathematical set A it belongs to (and its potential inverse). For example, the number 5 is not invertible in the set $\mathbb{Z}$ of integers, since its inverse $1/5$ is not an integer:

```
sage: a = 5; a
5
sage: a.is_unit()
False
```

However, it is invertible in the set of rational numbers:

```
sage: a = 5/1; a
5
sage: a.is_unit()
True
```

Sage gives two different answers to that question since, as seen in the above section, the objects 5 and $5/1$ have different classes.

In some object-oriented computer algebra systems, like MuPAD or Axiom, the mathematical set X to which x belongs (here $\mathbb{Z}$ or $\mathbb{Q}$) is simply the class of x. Sage follows the approach of the Magma system, and defines the set X by another object attached to x, called its *parent*:

```
sage: parent(5)
Integer Ring
sage: parent(5/1)
Rational Field
```

We can obtain these two sets with the following shortcuts:

```
sage: ZZ
Integer Ring
```

```
sage: QQ
Rational Field
```

and use them to easily *convert* an element from one set to the other, when it makes sense:

```
sage: QQ(5).parent()
Rational Field
sage: ZZ(5/1).parent()
Integer Ring
sage: ZZ(1/5)
Traceback (most recent call last):
  ...
TypeError: no conversion of this rational to integer
```

More generally, the `P(x)` syntax — where `P` is a parent — tries to convert the object `x` into an element of `P`. We show four different instances of 1: as integer $1 \in \mathbb{Z}$, as rational number $1 \in \mathbb{Q}$, as real floating-point $1.0 \in \mathbb{R}$ or complex floating-point $1.0 + 0.0i \in \mathbb{C}$:

```
sage: ZZ(1), QQ(1), RR(1), CC(1)
(1, 1, 1.00000000000000, 1.00000000000000)
```

Exercise 19. Find two Sage objects having the same type and different parents. Then find two Sage objects having the same parent and different types.

5.2.2 Constructions

The parents being themselves first-class objects, we can apply operations to them. For example, one can construct the cartesian product $\mathbb{Q}^2$:

```
sage: cartesian_product([QQ, QQ])
The Cartesian product of (Rational Field, Rational Field)
```

find $\mathbb{Q}$ as the fraction field of $\mathbb{Z}$:

```
sage: ZZ.fraction_field()
Rational Field
```

construct the ring of polynomials in x with coefficients in $\mathbb{Z}$:

```
sage: ZZ['x']
Univariate Polynomial Ring in x over Integer Ring
```

Using an incremental approach, we can construct complex algebraic structures like the 3×3 matrix space with polynomial coefficients on a finite field:

```
sage: Z5 = GF(5); Z5
Finite Field of size 5
sage: P = Z5['x']; P
Univariate Polynomial Ring in x over Finite Field of size 5
sage: M = MatrixSpace(P, 3, 3); M
```

```
Full MatrixSpace of 3 by 3 dense matrices over
Univariate Polynomial Ring in x over Finite Field of size 5
```

and draw a random element from this domain:

```
sage: M.random_element()
[2*x^2 + 3*x + 4 4*x^2 + 2*x + 2     4*x^2 + 2*x]
[           3*x   2*x^2 + x + 3     3*x^2 + 4*x]
[     4*x^2 + 3 3*x^2 + 2*x + 4         2*x + 4]
```

5.2.3 Further Reading: Categories

In general, a parent does not itself have a parent, but a *category* that indicates its properties:

```
sage: QQ.category()
Join of Category of number fields and Category of quotient fields and
    Category of metric spaces
```

Sage knows that $\mathbb{Q}$ is a field:

```
sage: QQ in Fields()
True
```

and thus, for instance, an additive and commutative group (see Figure 5.1):

```
sage: QQ in CommutativeAdditiveGroups()
True
```

Since $\mathbb{Q}$ is a field, $\mathbb{Q}[x]$ is a Euclidean ring:

```
sage: QQ['x'] in EuclideanDomains()
True
```

All these properties are used to provide rigorous and efficient computations on elements of these sets.

5.3 Domains with a Normal Form

Let us now browse some of the parents we will encounter in Sage.

We have seen in §2.1 how important normal forms[2] can be in computer algebra, since they allow to determine if two objects are mathematically equal in comparing their normal form representations. Each of the fundamental parents presented in this section corresponds to a *domain with normal form*, i.e., a set of mathematical objects having a normal form. This allows Sage to represent without any ambiguity the elements of each of these parents[3].

[2] In this book we use both *canonical form* and *normal form* to mean that two objects are mathematically identical if their canonical (or normal) forms are equal. Sometimes *normal form* is meant as a weaker notion, where only zero is assumed to have a unique representation.

[3] Most of the other parents available in Sage correspond to domains with a normal form, but not all of them. It also happens that, for efficiency reasons, Sage *represents* elements in normal form only when explicitly requested.

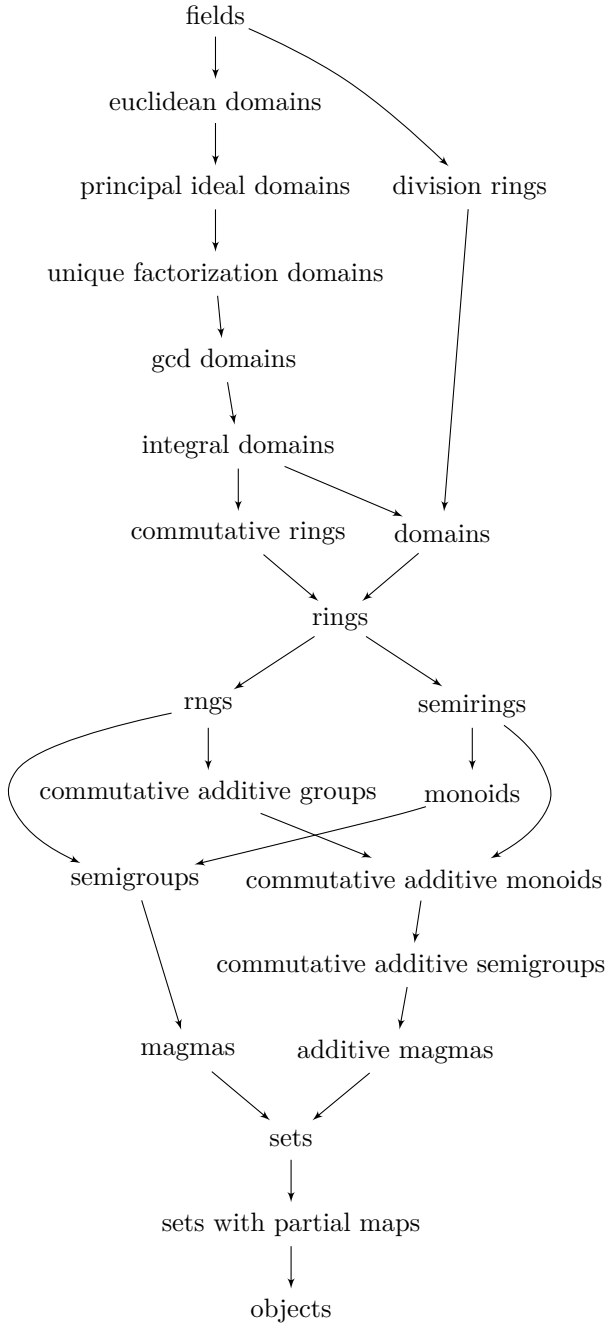

FIGURE 5.1 – A short part of the category graph in Sage.

Some basic Python types	
Python integers	`int`
Python floating-point numbers	`float`
Booleans (*true*, *false*)	`bool`
Character strings	`str`

Basic numerical domains	
Integers $\mathbb{Z}$	`ZZ` *or* `IntegerRing()`
Rational numbers $\mathbb{Q}$	`QQ` *or* `RationalField()`
Floating-point numbers with p bits	`Reals(p)` *or* `RealField(p)`
Complex floating-point numbers with p bits	`Complexes(p)` *or* `ComplexField(p)`

Rings and finite fields	
Integers modulo n, $\mathbb{Z}/n\mathbb{Z}$	`Integers(n)` *or* `IntegerModRing(n)`
Finite field $\mathbb{F}_q$	`GF(q)` *or* `FiniteField(q)`

Algebraic numbers	
Algebraic numbers $\bar{\mathbb{Q}}$	`QQbar` *or* `AlgebraicField()`
Real algebraic numbers	`AA` *or* `AlgebraicRealField()`
Number fields $\mathbb{Q}[x]/\langle p \rangle$	`NumberField(p)`

Symbolic computation	
Matrices $m \times n$ with coefficients in A	`MatrixSpace(A, m, n)`
Polynomials $A[x, y]$	`A['x,y']` *or* `PolynomialRing(A, 'x,y')`
Series $A[[x]]$	`A[['x']]` *or* `PowerSeriesRing(A, 'x')`
Symbolic expressions	`SR`

TABLE 5.1 – Main domains and parents.

5.3.1 Elementary Domains

We call *elementary computational domains* (or simply elementary domains) the classical sets of constants, with no variable: integers, rational numbers, floating-point numbers, booleans, integers modulo n...

Integers. The integers are represented in radix two internally, and printed by default in radix ten. As seen above, the Sage integers are objects of the class `Integer`. Their parent is the ring $\mathbb{Z}$:

```
sage: 5.parent()
Integer Ring
```

The integers are always in normal form; their equality is thus easy to check. As a consequence, to be able to represent integers in factorised form, the `factor` command needs a specific class:

```
sage: type(factor(4))
<class 'sage.structure.factorization_integer.IntegerFactorization'>
```

The `Integer` class is specific to Sage: by default, Python uses integers of type `int`. In general, the conversion from `Integer` to `int` — or vice versa — is automatic, but it might be necessary to convert explicitly by

```
sage: int(5)
5
sage: type(int(5))
<type 'int'>
```

or conversely

```
sage: Integer(5)
5
sage: type(Integer(5))
<type 'sage.rings.integer.Integer'>
```

Rational Numbers. The normal form property extends to rational numbers, elements of `QQ`, which are always represented in reduced form. Therefore, in the command

```
sage: factorial(99) / factorial(100) - 1 / 50
-1/100
```

the factorials are first evaluated, then the obtained fraction 1/100 is put into reduced form. Sage then constructs the rational number 1/50, performs the subtraction, then reduces again the result (there is nothing to do here).

Floating-Point Numbers. Real numbers cannot all be exactly represented in a finite format. Their numerical values are approximated by floating-point numbers, which will be discussed in more detail in Chapter 11.

Within Sage, floating-point numbers are encoded in binary radix. As a consequence, the floating-point number corresponding to the input `0.1` slightly differs from 1/10, since 1/10 is not exactly representable in binary! Each floating-point number has its own precision. The parent of floating-point numbers with p-bit significand is denoted `Reals(p)`, which for the default precision ($p = 53$) is also denoted `RR`. As for integers, Sage floating-point numbers differ from their Python analogue.

When they appear in a sum, product or quotient containing also integers or rational numbers, floating-point numbers are "contagious"; the complete expression is then evaluated as a floating-point number:

```
sage: 72/53 - 5/3 * 2.7
-3.14150943396227
```

Likewise, when the argument of some usual function is a floating-point number, the result is again a floating-point number:

```
sage: cos(1), cos(1.)
(cos(1), 0.540302305868140)
```

The `numerical_approx` method (or its alias `n`) evaluates numerically the remaining expressions. An optional argument allows us to set the number of significant digits used for this evaluation. Here is for example π with 50 significant digits:

```
sage: pi.n(digits=50)    # variant: n(pi,digits=50)
3.1415926535897932384626433832795028841971693993751
```

Complex Floating-Point Numbers. Similarly, the floating-point approximations of complex numbers with precision p are elements of `Complexes(p)` — or its alias `ComplexField(p)` —, or `CC` with the default precision of 53 bits. For example, we can construct a complex floating-point number and compute its argument by

```
sage: z = CC(1,2); z.arg()
1.10714871779409
```

Complex symbolic expressions

The imaginary unit i (denoted `I` or `i`), already encountered in the preceding chapters, is not an element of `CC`, but a symbolic expression (see §5.4.1):

```
sage: I.parent()
Symbolic Ring
```

We can use it to define a complex floating-point number with an explicit conversion:

```
sage: (1.+2.*I).parent()
Symbolic Ring
sage: CC(1.+2.*I).parent()
Complex Field with 53 bits of precision
```

In the world of symbolic expressions, the methods `real`, `imag` and `abs` give respectively the real part, the imaginary part and the modulus of a complex number:

```
sage: z = 3 * exp(I*pi/4)
sage: z.real(), z.imag(), z.abs().canonicalize_radical()
(3/2*sqrt(2), 3/2*sqrt(2), 3)
```

Booleans. Logic expressions also form a computational domain with normal form, but the class of boolean values is a basic type without specific parent in Sage. The two normal forms are `True` and `False` (or `true` and `false`):

```
sage: a, b, c = 0, 2, 3
sage: a == 1 or (b == 2 and c == 3)
True
```

In tests and loops, the conditions built from the operators `or` and `and` are evaluated lazily from left to right. This means that the evaluation of a condition `or` ends as soon as the first `True` value is encountered, without evaluating the rightmost terms; similarly with `and` and `False`. Hence the following divisibility test of b by a does not produce any error even if $a = 0$:

```
sage: a = 0; b = 12; (a == 0 and b == 0) or (a != 0 and b % a == 0)
```

The operator `not` takes precedence over `and`, which in turn takes precedence over `or`, the equality and comparison tests having precedence over all boolean operators. The two following tests are thus equivalent to the above one:

```
sage: ((a == 0) and (b == 0)) or ((a != 0) and (b % a == 0))
sage: a == 0 and b == 0 or not a == 0 and b % a == 0
```

In addition, Sage allows multiple equality or inequality tests, exactly like in mathematics:

$x \leq y < z \leq t$	encoded by	`x <= y < z <= t`
$x = y = z \neq t$		`x == y == z != t`

In the simple cases, these tests are automatically performed; otherwise we call the `bool` command to force the evaluation:

```
sage: x, y = var('x, y')
sage: bool( (x-y)*(x+y) == x^2-y^2 )
True
```

Integers Modulo n. To define an integer modulo n, we first build its parent, the ring $\mathbb{Z}/n\mathbb{Z}$:

```
sage: Z4 = IntegerModRing(4); Z4
Ring of integers modulo 4
sage: m = Z4(7); m
3
```

As in the case of floating-point numbers, the computations involving m are done modulo 4 via automatic conversions. In the following example, 3 and 1 are automatically converted in elements of $\mathbb{Z}/4\mathbb{Z}$:

```
sage: 3 * m + 1
2
```

When p is prime, we can also choose to build $\mathbb{Z}/p\mathbb{Z}$ as a field:

```
sage: Z3 = GF(3); Z3
Finite Field of size 3
```

Both `IntegerModRing(n)` and `GF(p)` are domains with a normal form: the reduction modulo n or p are done automatically. The computations in rings and finite fields are detailed in Chapter 6.

5.3.2 Compound Domains

From well-defined constants, some classes of symbolic objects with variables and having a normal form can be constructed. The most important such classes are matrices, polynomials, rational functions and truncated power series.

The corresponding parents are parameterised by their coefficient domain. For example, matrices with integer coefficients differ from matrices with coefficients in $\mathbb{Z}/n\mathbb{Z}$, and the corresponding computation rules are automatically applied, without requiring an explicit call to a function reducing integers modulo n.

Part II of this book is mainly dedicated to these objects.

Matrices. The normal form[4] of a matrix is obtained when all its coefficients are themselves in normal form. As a consequence, a matrix defined over a field or ring with normal form is automatically in normal form:

```
sage: a = matrix(QQ, [[1,2,3],[2,4,8],[3,9,27]])
sage: (a^2 + 1) * a^(-1)
[  -5 13/2  7/3]
[   7    1 25/3]
[   2 19/2   27]
```

The `matrix` function call is a shortcut. Internally, Sage builds the corresponding parent, here the space of 3×3 matrices with coefficients in $\mathbb{Q}$ (which has normal form), then uses it to construct the matrix:

```
sage: M = MatrixSpace(QQ,3,3); M
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
sage: a = M([[1,2,3],[2,4,8],[3,9,27]])
sage: (a^2 + 1) * a^(-1)
[  -5 13/2  7/3]
[   7    1 25/3]
[   2 19/2   27]
```

The operations on symbolic matrices are described in Chapter 8, and on numerical matrices in Chapter 13.

Polynomials and Fractions. Like matrices, polynomials in Sage "know" the type of their coefficients. Their parents are polynomial rings like $\mathbb{Z}[x]$ or $\mathbb{C}[x, y, z]$, presented in detail in Chapters 7 and 9, and which can be built as follows:

```
sage: P = ZZ['x']; P
Univariate Polynomial Ring in x over Integer Ring
sage: F = P.fraction_field(); F
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: p = P(x+1) * P(x); p
x^2 + x
sage: p + 1/p
```

[4] Do not confuse this concept of normal form with the normal forms of a matrix viewed as a linear transformation, which will be discussed in Chapter 8.

```
(x^4 + 2*x^3 + x^2 + 1)/(x^2 + x)
sage: parent(p + 1/p)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

As we will see in §5.4.2, there is no optimal representation for polynomials and fractions. The elements of polynomial rings are represented in expanded form. These rings do therefore have a normal form as soon as the coefficients themselves belong to a domain with normal form.

These polynomials differ from the polynomial expressions (`Symbolic Ring`) we have seen in Chapter 2, which do not have a well-defined coefficient type, neither a parent reflecting such a type. The latter give an alternative to "true" polynomials, which can be useful, for example, to mix polynomials and other mathematical expressions. However, contrary to polynomial rings, when we work with such expressions, we have to explicitly call a *reduction command* like `expand` to put them in normal form (if such a form exists).

Power Series. Truncated power series are objects of the form

$$a_0 + a_1\,x + a_2\,x^2 + \cdots + a_n\,x^n + \mathcal{O}(x^{n+1})$$

used for example to represent Taylor expansions, and whose usage in Sage is described in §7.5. The parent of series in x, truncated at order n, and with coefficients in A, is the ring $A[[x]]$, build with `PowerSeriesRing(A, 'x', n)`.

Like polynomials, truncated power series have an analogue in the world `SR` of symbolic expressions. The corresponding command to reduce to normal form is `series`.

```
sage: f = cos(x).series(x == 0, 6); 1 / f
```

$$\frac{1}{1+(-\frac{1}{2})x^2+\frac{1}{24}x^4+O(x^6)}$$

```
sage: (1 / f).series(x == 0, 6)
```

$$1 + \tfrac{1}{2}x^2 + \tfrac{5}{24}x^4 + O\left(x^6\right)$$

Algebraic Numbers. An algebraic number is defined as root of a polynomial. When the polynomial degree is 5 or more, in general it is not possible to explicitly write its roots in terms of the operations $+, -, \times, /, \sqrt{\cdot}$. However, many computations involving the roots can be performed successfully without any other information than the polynomial itself.

```
sage: k.<a> = NumberField(x^3 + x + 1); a^3; a^4+3*a
-a - 1
-a^2 + 2*a
```

This book does not describe in detail how to play with algebraic numbers in Sage, however several examples can be found in Chapters 7 and 9.

5.4 Expressions vs Computational Domains

Several approaches are thus possible for manipulating objects like polynomials within Sage. We can consider them as particular symbolic expressions, as in the first chapters, or introduce a given ring of polynomials and compute with its elements. To conclude this chapter, we briefly describe the parent of symbolic expressions, the `SR` domain, then we demonstrate through several examples how important it is to control the domain of computations, and the differences between both approaches.

5.4.1 Symbolic Expressions as a Computational Domain

Symbolic expressions themselves form a computational domain. In Sage, their parent is the *symbolic ring*:

```
sage: parent(sin(x))
Symbolic Ring
```

that can also be obtained with:

```
sage: SR
Symbolic Ring
```

The properties of this ring are rather fuzzy; it is commutative:

```
sage: SR.category()
Category of commutative rings
```

and the computation rules assume roughly speaking that all symbolic variables are in $\mathbb{C}$.

The form of expressions in `SR` (polynomials, fractions, trigonometric expressions) being not apparent in their class or parent, the result of a computation often requires some manual transformations to obtain the desired form (see §2.1), by using for example `expand`, `combine`, `collect` and `simplify`. To use these functions well we have to know which kind of transformation they perform, to which sub-classes[5] of symbolic expressions these transformations apply, and which of these sub-classes have a normal form. In particular, the blind use of the `simplify` command can yield wrong results. Some variants of `simplify` allow then to precisely describe the transformation to apply.

5.4.2 Examples: Polynomials and Normal Forms

Let us build the ring $\mathbb{Q}[x_1, x_2, x_3, x_4]$ of polynomials in 4 variables:

```
sage: R = QQ['x1,x2,x3,x4']; R
Multivariate Polynomial Ring in x1, x2, x3, x4 over Rational Field
sage: x1, x2, x3, x4 = R.gens()
```

The elements of R are automatically put in expanded form:

[5] In the sense of subset, and not of Python class.

```
sage: x1 * (x2 - x3)
x1*x2 - x1*x3
```

which, as we have seen, is a normal form. In particular, the test to zero in R is trivial:

```
sage: (x1+x2)*(x1-x2) - (x1^2 - x2^2)
0
```

An expanded form is not always optimal. For example, if we build the Vandermonde determinant $\prod_{1\leq i<j\leq n}(x_i - x_j)$:

```
sage: prod( (a-b) for (a,b) in Subsets([x1,x2,x3,x4],2) )
x1^3*x2^2*x3 - x1^2*x2^3*x3 - x1^3*x2*x3^2 + x1*x2^3*x3^2
+ x1^2*x2*x3^3 - x1*x2^2*x3^3 - x1^3*x2^2*x4 + x1^2*x2^3*x4
+ x1^3*x3^2*x4 - x2^3*x3^2*x4 - x1^2*x3^3*x4 + x2^2*x3^3*x4
+ x1^3*x2*x4^2 - x1*x2^3*x4^2 - x1^3*x3*x4^2 + x2^3*x3*x4^2
+ x1*x3^3*x4^2 - x2*x3^3*x4^2 - x1^2*x2*x4^3 + x1*x2^2*x4^3
+ x1^2*x3*x4^3 - x2^2*x3*x4^3 - x1*x3^2*x4^3 + x2*x3^2*x4^3
```

we obtain $4! = 24$ terms. The same construct with an expression from `SR` remains under factored form, and is much more compact and readable:

```
sage: x1, x2, x3, x4 = SR.var('x1, x2, x3, x4')
sage: prod( (a-b) for (a,b) in Subsets([x1,x2,x3,x4],2) )
-(x1 - x2)*(x1 - x3)*(x1 - x4)*(x2 - x3)*(x2 - x4)*(x3 - x4)
```

In addition, a factored representation allows faster gcd computations. However, it would be unwise to put automatically every polynomial into factored form, even if this is also a normal form, since the factorisation is computationally expensive, and makes additions costly.

In general, depending on the kind of computation, the optimal representation of an element is not always its normal form (if it exists). This leads computer algebra systems to a compromise with expressions. Some basic simplifications, like the reduction of fractions or the multiplication by zero, are done automatically; the other transformations are left to the user with the provided specialised commands.

5.4.3 Example: Polynomial Factorisation

Let us consider the factorisation of the following polynomial expression:

```
sage: x = var('x')
sage: p = 54*x^4+36*x^3-102*x^2-72*x-12
sage: factor(p)
6*(x^2 - 2)*(3*x + 1)^2
```

Is this answer satisfying? It is indeed a factorisation of p, however its completeness heavily depends on the context! For now, Sage considers `p` as a symbolic expression, which happens to be polynomial. Sage cannot know if we wish to factor p as a

product of polynomials with integer coefficients, or with rational coefficients (for example).

To take full control, we will make it clear in which mathematical set (i.e., computational domain) p lives. To start, let us consider p as a polynomial with integer coefficients. We thus define the ring $R = \mathbb{Z}[x]$ of these polynomials:

```
sage: R = ZZ['x']; R
Univariate Polynomial Ring in x over Integer Ring
```

Then we convert p in this ring:

```
sage: q = R(p); q
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
```

The output seems identical, however q knows it is an element of R:

```
sage: parent(q)
Univariate Polynomial Ring in x over Integer Ring
```

As a consequence, its factorisation is uniquely defined:

```
sage: factor(q)
2 * 3 * (3*x + 1)^2 * (x^2 - 2)
```

Let us proceed similarly in the rational field:

```
sage: R = QQ['x']; R
Univariate Polynomial Ring in x over Rational Field
sage: q = R(p); q
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: factor(q)
(54) * (x + 1/3)^2 * (x^2 - 2)
```

In this new context, the factorisation is again well-defined, but different from the previous one.

Let us now compute a complete factorisation over the complex numbers. A first solution is to allow a numerical approximation of complex numbers with 16 bits of precision:

```
sage: R = ComplexField(16)['x']; R
Univariate Polynomial Ring in x over Complex Field
with 16 bits of precision
sage: q = R(p); q
54.00*x^4 + 36.00*x^3 - 102.0*x^2 - 72.00*x - 12.00
sage: factor(q)
(54.00) * (x - 1.414) * (x + 0.3333)^2 * (x + 1.414)
```

Another solution is to extend the field of rational numbers, e.g., adding $\sqrt{2}$.

```
sage: R = QQ[sqrt(2)]['x']; R
Univariate Polynomial Ring in x over Number Field in sqrt2
with defining polynomial x^2 - 2
sage: q = R(p); q
```

```
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: factor(q)
(54) * (x - sqrt2) * (x + sqrt2) * (x + 1/3)^2
```

Finally, maybe we want that coefficients be considered modulo 5?

```
sage: R = GF(5)['x']; R
Univariate Polynomial Ring in x over Finite Field of size 5
sage: q = R(p); q
4*x^4 + x^3 + 3*x^2 + 3*x + 3
sage: factor(q)
(4) * (x + 2)^2 * (x^2 + 3)
```

5.4.4 Synthesis

In the preceding examples, we have shown how the user might control the level of rigour in her/his computations.

On the one hand, she/he can use symbolic expressions. These expressions live in the ring `SR`. They offer several methods (presented in Chapter 2) which apply well to some sub-classes of expressions, like polynomial expressions. When we recognise to which classes a given expression belongs to, this helps to know which functions could be applied. The simplification of expressions is a particular problem where this recognition is crucial. The main classes of expression are defined to take into account this simplification issue, and we will prefer this approach in the rest of this book.

On the other hand, the user can *construct* a parent which will explicitly define the computational domain. It is especially interesting when this parent has a *normal form*: i.e., when two objects are mathematically equal if and only if they have the same representation.

As a summary, the main advantage of symbolic expressions (`SR`) is their ease of use: no explicit declaration of the computational domain, easy addition of new variables or functions, easy change of the computational domain (for example when one takes the sine of a polynomial expression), use of all possible calculus tools (integration, etc.). The advantages of explicitly defining the computational domain are in the first place pedagogical, more rigorous computations[6], the automatic normal form transformation (which can also be a drawback!), and the easy access to advanced constructions that would be difficult with symbolic expressions (computations in a finite field or an algebraic extension of $\mathbb{Q}$, in a non-commutative ring, etc.).

[6]Sage is not a *certified* computer algebra system: a bug is thus always possible; however, there will be no use of implicit assumption.

Part II

Algebra and Symbolic Computation