Thematic Tutorials

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

This is an index, grouped by theme, of Sage demonstrations, quick reference cards, primers, and thematic tutorials, licensed under a Creative Commons Attribution-Share Alike 3.0 License.

- A *quickref* (or quick reference card) is a one page document with the essential examples, and pointers to the main entry points.
- A *primer* is a document meant for a user to get started by himself on a theme in a matter of minutes.
- A *tutorial* is more in-depth and could take as much as an hour or more to get through.

Some of those tutorials are part of a series developed for the MAA PREP Workshop Sage: Using Open-Source Mathematics Software with Undergraduates.

CHAPTER

ONE

INTRODUCTION TO SAGE

- Logging on to a Sage Server and Creating a Worksheet (PREP)
- Introductory Sage Tutorial (PREP)
- *Tutorial: Using the Sage notebook, navigating the help system, first exercises*
- Sage's main tutorial

See also the *Python and Sage programming tutorials* below.

CHAPTER

TWO

CALCULUS AND PLOTTING

- Tutorial: Symbolics and Plotting (PREP)
- Tutorial: Calculus (PREP)
- Tutorial: Advanced-2D Plotting (PREP)

CHAPTER

THREE

ALGEBRA

- *Linear Programming (Mixed Integer)*
- *Group Theory and Sage*
- *Lie Methods and Related Combinatorics in Sage*

3.1 Number Theory

- *Number Theory and the RSA Public Key Cryptosystem*
- *Introduction to the p-adics*

CHAPTER

FOUR

COMBINATORICS

- *Introduction to combinatorics in Sage*
- *Algebraic Combinatorics in Sage*

4.1 Algebraic Combinatorics

- `Tutorial: Symmetric Functions`
- *Lie Methods and Related Combinatorics in Sage*
- *Tutorial: visualizing root systems*
- *Abelian Sandpile Model*

CHAPTER

FIVE

PROGRAMMING AND DESIGN

- Tutorial: Sage Introductory Programming (PREP)
- *Tutorial: Programming in Python and Sage*
- *Tutorial: Comprehensions, Iterators, and Iterables*
- *Tutorial: Objects and Classes in Python and Sage*
- *Functional Programming for Mathematicians*

5.1 Modeling Mathematics on a computer

- *How to implement new algebraic structures in Sage*
- *Elements, parents, and categories in Sage: a (draft of) primer*
- *Implementing a new parent: a (draft of) tutorial*

CHAPTER

SIX

INDICES AND TABLES

- *search*

6.1 Thematic tutorial document tree

6.1.1 Algebraic Combinatorics in Sage

Author: Anne Schilling (UC Davis)

These notes provide some Sage examples for Stanley's book:

A free pdf version of the book without exercises can be found on Stanley's homepage.

Preparation of this document was supported in part by NSF grants DMS–1001256 and OCI–1147247.

I would like to thank Federico Castillo (who wrote a first version of the n-cube section) and Nicolas M. Thiery (who wrote a slightly different French version of the Tsetlin library section) for their help.

Walks in graphs

This section provides some examples on Chapter 1 of Stanley's book [Stanley2013].

We begin by creating a graph with 4 vertices:

```
sage: G = Graph(4)
sage: G
Graph on 4 vertices
```

This graph has no edges yet:

```
sage: G.vertices()
[0, 1, 2, 3]
sage: G.edges()
[]
```

Before we can add edges, we need to tell Sage that our graph can have loops and multiple edges.:

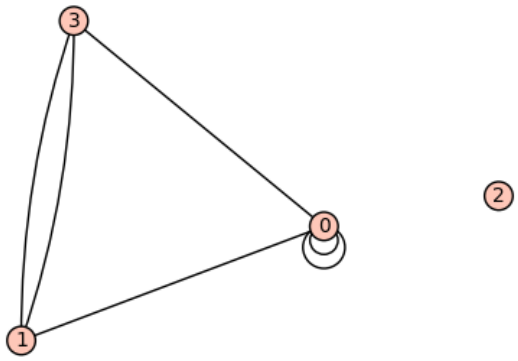
```
sage: G.allow_loops(True)
sage: G.allow_multiple_edges(True)
```

Now we are ready to add our edges by specifying a tuple of vertices that are connected by an edge. If there are multiple edges, we need to add the tuple with multiplicity.:

```
sage: G.add_edges([(0,0),(0,0),(0,1),(0,3),(1,3),(1,3)])
```

Now let us look at the graph!

```
sage: G.plot()
```



We can construct the adjacency matrix:

```
sage: A = G.adjacency_matrix()
sage: A
[2 1 0 1]
[1 0 0 2]
[0 0 0 0]
[1 2 0 0]
```

The entry in row i and column j of the ℓ-th power of A gives us the number of paths of length ℓ from vertex i to vertex j. Let us verify this:

```
sage: A**2
[6 4 0 4]
[4 5 0 1]
[0 0 0 0]
[4 1 0 5]
```

There are 4 paths of length 2 from vertex 0 to vertex 1: take either loop at 0 and then the edge $(0, 1)$ (2 choices) or take the edge $(0, 3)$ and then either of the two edges $(3, 1)$ (two choices):

```
sage: (A**2)[0,1]
4
```

To count the number of closed walks, we can also look at the sum of the ℓ-th powers of the eigenvalues. Even though the eigenvalues are not integers, we find that the sum of their squares is an integer:

```
sage: A.eigenvalues()
[0, -2, 0.5857864376269049?, 3.414213562373095?]
sage: sum(la**2 for la in A.eigenvalues())
16.000000000000000?
```

We can achieve the same by looking at the trace of the ℓ-th power of the matrix:

```
sage: (A**2).trace()
16
```

n-Cube

This section provides some examples on Chapter 2 of Stanley's book [Stanley2013], which deals with n-cubes, the Radon transform, and combinatorial formulas for walks on the n-cube.

The vertices of the n-cube can be described by vectors in $\mathbb{Z}_2^n$. First we define the addition of two vectors $u, v \in \mathbb{Z}_2^n$ via the following distance:

```
sage: def dist(u,v):
....:     h = [(u[i]+v[i])%2 for i in range(len(u))]
....:     return sum(h)
```

The distance function measures in how many slots two vectors in $\mathbb{Z}_2^n$ differ:

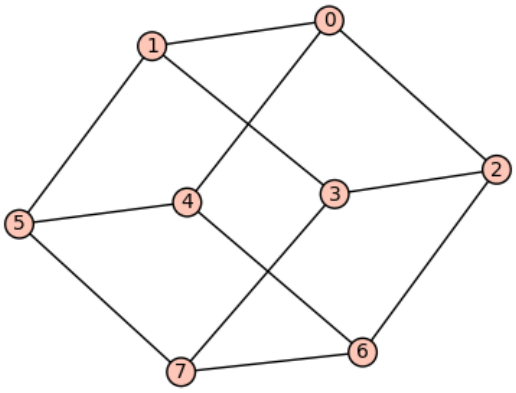
```
sage: u=(1,0,1,1,1,0)
sage: v=(0,0,1,1,0,0)
sage: dist(u,v)
2
```

Now we are going to define the n-cube as the graph with vertices in $\mathbb{Z}_2^n$ and edges between vertex u and vertex v if they differ in one slot, that is, the distance function is 1:

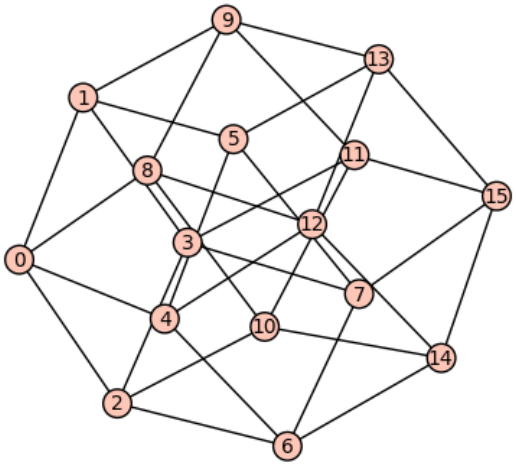
```
sage: def cube(n):
....:     G = Graph(2**n)
....:     vertices = Tuples([0,1],n)
....:     for i in range(2**n):
....:         for j in range(2**n):
....:             if dist(vertices[i],vertices[j]) == 1:
....:                 G.add_edge(i,j)
....:     return G
```

We can plot the 3 and 4-cube:

```
sage: cube(3).plot()
```



```
sage: cube(4).plot()
```



Next we can experiment and check Corollary 2.4 in Stanley's book, which states the n-cube has n choose i eigenvalues equal to $n - 2i$:

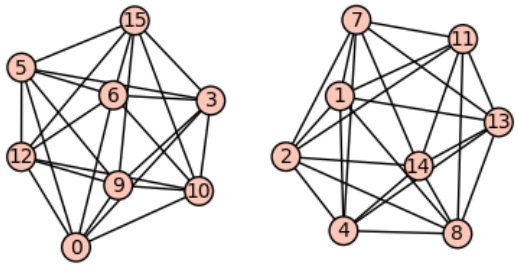
```
sage: G = cube(2)
sage: G.adjacency_matrix().eigenvalues()
[2, -2, 0, 0]

sage: G = cube(3)
sage: G.adjacency_matrix().eigenvalues()
[3, -3, 1, 1, 1, -1, -1, -1]

sage: G = cube(4)
sage: G.adjacency_matrix().eigenvalues()
[4, -4, 2, 2, 2, 2, -2, -2, -2, -2, 0, 0, 0, 0, 0, 0]
```

It is now easy to slightly vary this problem and change the edge set by connecting vertices u and v if their distance is 2 (see Problem 4 in Chapter 2):

```
sage: def cube_2(n):
....:     G = Graph(2**n)
....:     vertices = Tuples([0,1],n)
....:     for i in range(2**n):
....:         for j in range(2**n):
....:             if dist(vertices[i],vertices[j]) == 2:
....:                 G.add_edge(i,j)
....:     return G

sage: G = cube_2(2)
sage: G.adjacency_matrix().eigenvalues()
[1, 1, -1, -1]

sage: G = cube_2(4)
sage: G.adjacency_matrix().eigenvalues()
[6, 6, -2, -2, -2, -2, -2, -2, 0, 0, 0, 0, 0, 0, 0, 0]
```

Note that the graph is in fact disconnected. Do you understand why?

```
sage: cube_2(4).plot()
```



The Tsetlin library

Introduction

In this section, we study a simple random walk (or Markov chain), called the *Tsetlin library*. It will give us the opportunity to see the interplay between combinatorics, linear algebra, representation theory and computer exploration, without requiring heavy theoretical background. I hope this encourages everyone to play around with this or similar systems and investigate their properties! Formal theorems and proofs can be found in the references at the end of this section.

It has been known for several years that the theory of group representations can facilitate the study of systems whose evolution is random (Markov chains), breaking them down into simpler systems. More recently it was realized that generalizing this (namely replacing the invertibility axiom for groups by other axioms) explains the behavior of other particularly simple Markov chains such as the Tsetlin library.

The Tsetlin library

Consider a bookshelf in a library containing n distinct books. When a person borrows a book and then returns it, it gets placed back on the shelf to the right of all books. This is what we naturally do with our pile of shirts in the closet: after use and cleaning, the shirt is placed on the top of its pile. Hence the most popular books/shirts will more likely appear on the right/top of the shelf/pile.

This type of organization has the advantage of being self-adaptive:

- The books most often used accumulate on the right and thus can easily be found.
- If the use changes over time, the system adapts.

In fact, this type of strategy is used not only in everyday life, but also in computer science. The natural questions that arise are:

- *Stationary distribution*: To which state(s) does the system converge to? This, among other things, is used to evaluate the average access time to a book.
- *The rate of convergence*: How fast does the system adapt to a changing environment .

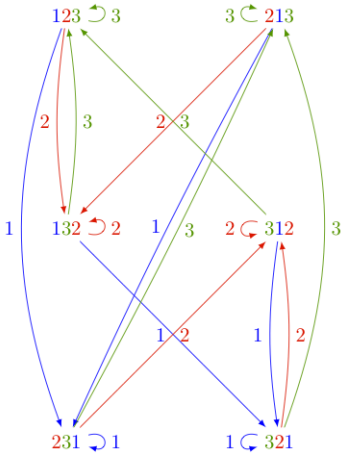
Let us formalize the description. The Tsetlin library is a discrete Markov chain (discrete time, discrete state space) described by:

- The state space Ω_n is given by the set of all permutations of the n books.
- The transition operators are denoted by $\partial_i \colon \Omega_n \to \Omega_n$. When ∂_i is applied to a permutation σ, the number i is moved to the end of the permutation.

- We assign parameters $x_i \geq 0$ for all $1 \leq i \leq n$ with $\sum_{i=1}^{n} x_i = 1$. The parameter x_i indicates the probability of choosing the operator ∂_i.

Transition graph and matrix

One can depict the action of the operators ∂_i on the state space Ω_n by a digraph. The following picture shows the action of $\partial_1, \partial_2, \partial_3$ on Ω_3:



The above picture can be reproduced in Sage as follows:

```
sage: P = Poset(([1,2,3],[]))
```

This is the antichain poset. Its linear extensions are all permutations of $\{1, 2, 3\}$:

```
sage: L = P.linear_extensions()
sage: L
The set of all linear extensions of Finite poset containing 3 elements
sage: L.list()
[[3, 2, 1], [3, 1, 2], [2, 3, 1], [2, 1, 3], [1, 3, 2], [1, 2, 3]]
```

The graph is produced via:

```
sage: G = L.markov_chain_digraph(labeling='source'); G
Looped multi-digraph on 6 vertices
sage: view(G) # optional - dot2tex
```

We can now look at the transition matrix and see whether we notice anything about its eigenvalue and eigenvectors:

```
sage: M = L.markov_chain_transition_matrix(labeling='source')
sage: M
[-x1 - x2       x0        0        0       x0        0]
[      x1 -x0 - x2       x1        0        0        0]
[       0        0 -x1 - x2       x0        0       x0]
[      x2        0       x2 -x0 - x1        0        0]
[       0        0        0       x1 -x0 - x2       x1]
[       0       x2        0        0       x2 -x0 - x1]
```

This matrix is normalized so that all columns add to 0. So we need to add $(x_0 + x_1 + x_2)$ times the 6×6 identity matrix to get the probability matrix:

```
sage: x = M.base_ring().gens()
sage: Mt = (x[0]+x[1]+x[2])*matrix.identity(6)+M
sage: Mt
[x0 x0  0  0 x0  0]
[x1 x1 x1  0  0  0]
[ 0  0 x0 x0  0 x0]
[x2  0 x2 x2  0  0]
[ 0  0  0 x1 x1 x1]
[ 0 x2  0  0 x2 x2]
```

Since the x_i are formal variables, we need to compute the eigenvalues and eigenvectors in the symbolic ring `SR`:

```
sage: Mt.change_ring(SR).eigenvalues()
[x2, x1, x0, x0 + x1 + x2, 0, 0]
```

Do you see any pattern? In fact, if you start playing with bigger values of n (the size of the underlying permutations), you might observe that there is an eigenvalue for every subset S of $\{1, 2, \ldots, n\}$ and the multiplicity is given by a derangement number $d_{n-|S|}$. Derangment numbers count permutations without fixed point. For the eigenvectors we obtain:

```
sage: Mt.change_ring(SR).eigenvectors_right()
[(x2, [(0, 0, 0, 1, 0, -1)], 1),
 (x1, [(0, 1, 0, 0, -1, 0)], 1),
 (x0, [(1, 0, -1, 0, 0, 0)], 1),
 (x0 + x1 + x2,
 [(1, (x1 + x2)/(x0 + x2), x2/x1, (x1*x2 + x2^2)/(x0*x1 + x1^2),
  (x1*x2 + x2^2)/(x0^2 + x0*x2), (x1*x2 + x2^2)/(x0^2 + x0*x1))], 1),
  (0, [(1, 0, -1, 0, -1, 1), (0, 1, -1, 1, -1, 0)], 2)]
```

The stationary distribution is the eigenvector of eigenvalues $1 = x_0 + x_1 + x_2$. Do you see a pattern?

Optional exercices: Study of the transition operators and graph

Instead of using the methods that are already in Sage, try to build the state space Ω_n and the transition operators ∂_i yourself as follows.

1. For technical reasons, it is most practical in Sage to label the n books in the library by $0, 1, \cdots, n-1$, and to represent each state in the Markov chain by a permutation of the set $\{0, \ldots, n-1\}$ as a tuple. Construct the state space Ω_n as:

   ```
   sage: map(tuple, Permutations(range(3)))
   [(0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 0, 1), (2, 1, 0)]
   ```

2. Write a function `transition_operator(sigma, i)` which implements the operator ∂_i which takes as input a tuple `sigma` and integer $i \in \{1, 2, \ldots, n\}$ and outputs a new tuple. It might be useful to extract subtuples (`sigma[i:j]`) and concatentation.
3. Write a function `tsetlin_digraph(n)` which constructs the (multi digraph) as described as shown above. This can be achieved using `DiGraph`.
4. Verify for which values of n the digraph is strongly connected (i.e., you can go from any vertex to any other vertex by going in the direction of the arrow). This indicates whether the Markov chain is irreducible.

Conclusion

The Tsetlin library was studied from the viewpoint of monoids in [Bidigare1997] and [Brown2000]. Precise statements of the eigenvalues and the stationary distribution of the probability matrix as well as proofs of the statements are given in these papers. Generalizations of the Tsetlin library from the antichain to arbitrary posets was given in [AKS2013].

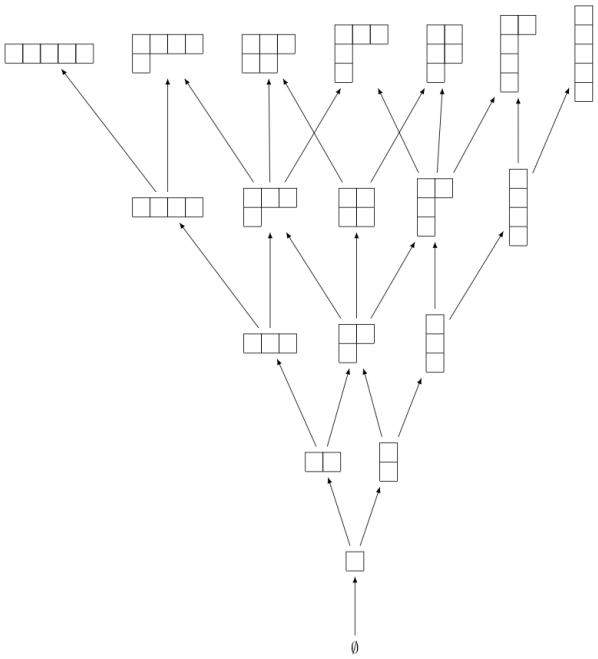
Young's lattice and the RSK algorithm

This section provides some examples on Young's lattice and the RSK (Robinson-Schensted-Knuth) algorithm explained in Chapter 8 of Stanley's book [Stanley2013].

Young's Lattice

We begin by creating the first few levels of Young's lattice Y. For this, we need to define the elements and the order relation for the poset, which is containment of partitions:

```
sage: level = 6
sage: elements = [b for n in range(level) for b in Partitions(n)]
sage: ord = lambda x,y: y.contains(x)
sage: Y = Poset((elements,ord), facade=True)
sage: H = Y.hasse_diagram()
sage: view(H)       # optional - dot2tex
```



We can now define the up and down operators U and D on $\mathbf{Q}Y$. First we do so on partitions, which form a basis for $\mathbf{Q}Y$:

```
sage: QQY = CombinatorialFreeModule(QQ,elements)

sage: def U_on_basis(la):
....:     covers = Y.upper_covers(la)
....:     return QQY.sum_of_monomials(covers)

sage: def D_on_basis(la):
....:     covers = Y.lower_covers(la)
....:     return QQY.sum_of_monomials(covers)
```

As a shorthand, one also can write the above as:

```
sage: U_on_basis = QQY.sum_of_monomials * Y.upper_covers
sage: D_on_basis = QQY.sum_of_monomials * Y.lower_covers
```

Here is the result when we apply the operators to the partition $(2,1)$:

```
sage: la = Partition([2,1])
sage: U_on_basis(la)
B[[2, 1, 1]] + B[[2, 2]] + B[[3, 1]]
sage: D_on_basis(la)
B[[1, 1]] + B[[2]]
```

Now we define the up and down operator on $\mathbf{Q}Y$:

```
sage: U = QQY.module_morphism(U_on_basis)
sage: D = QQY.module_morphism(D_on_basis)
```

We can check the identity $D_{i+1}U_i - U_{i-1}D_i = I_i$ explicitly on all partitions of $i = 3$:

```
sage: for p in Partitions(3):
....:     b = QQY(p)
....:     assert D(U(b)) - U(D(b)) == b
```

We can also check that the coefficient of $\lambda \vdash n$ in $U^n(\emptyset)$ is equal to the number of standard Young tableaux of shape λ:

```
sage: u = QQY(Partition([]))
sage: for i in range(4):
....:     u = U(u)
sage: u
B[[1, 1, 1]] + 3*B[[2, 1, 1]] + 2*B[[2, 2]] + 3*B[[3, 1]] + B[[4]]
```

For example, the number of standard Young tableaux of shape $(2,1,1)$ is 3:

```
sage: StandardTableaux([2,1,1]).cardinality()
3
```

We can test this in general:

```
sage: for la in u.support():
....:     assert u[la] == StandardTableaux(la).cardinality()
```

We can also check this against the hook length formula (Theorem 8.1):

```
sage: def hook_length_formula(p):
....:     n = p.size()
....:     return factorial(n) / prod(p.hook_length(*c) for c in p.cells())

sage: for la in u.support():
....:     assert u[la] == hook_length_formula(la)
```

RSK Algorithm

Let us now turn to the RSK algorithm. We can verify Example 8.12 as follows:

```
sage: p = Permutation([4,2,7,3,6,1,5])
sage: RSK(p)
[[[1, 3, 5], [2, 6], [4, 7]], [[1, 3, 5], [2, 4], [6, 7]]]
```

The tableaux can also be displayed as tableaux:

```
sage: P,Q = RSK(p)
sage: P.pp()
1  3  5
2  6
4  7
sage: Q.pp()
1  3  5
2  4
6  7
```

The inverse RSK algorithm is implemented as follows:

```
sage: RSK_inverse(P,Q, output='permutation')
[4, 2, 7, 3, 6, 1, 5]
```

We can verify that the RSK algorithm is a bijection:

```
sage: def check_RSK(n):
....:     for p in Permutations(n):
....:         assert RSK_inverse(*RSK(p), output='permutation') == p
sage: for n in range(5):
....:     check_RSK(n)
```

6.1.2 Tutorial: Using the Sage notebook, navigating the help system, first exercises

This worksheet is based on William Stein's JPL09__intro_to_sage.sws worksheet and the Sage days 20.5_demo worksheet and aims to be an interactive introduction to Sage through exercises. You will learn how to use the notebook and call the help.

Making this help page into a worksheet

If you are browsing this document as a static web page, you can see all the examples; however you need to copy-paste them one by one to experiment with them. Use the `Upload worksheet` button of the notebook and copy-paste the URL of this page to obtain an editable copy in your notebook.

If you are browsing this document as part of Sage's live documentation, you can play with the examples directly here; however your changes will be lost when you close this page. Use `Copy worksheet` from the `File...` menu at the top of this page to get an editable copy in your notebook.

Both in the live tutorial and in the notebook, you can clear all output by selecting `Delete All Output` from the `Action...` menu next to the `File...` menu at the top of the worksheet.

Entering, Editing and Evaluating Input

To *evaluate code* in the Sage Notebook, type the code into an input cell and press `shift-enter` or click the `evaluate` link. Try it now with a simple expression (e.g., $2+3$). The first time you evaluate a cell takes longer than subsequent times since a new Sage process is started:

```
sage: 2 + 3
5

sage: # edit here

sage: # edit here
```

To create *new input cells*, click the blue line that appears between cells when you move your mouse around. Try it now:

```
sage: 1 + 1
2

sage: # edit here
```

You can *go back* and edit any cell by clicking in it (or using the arrow keys on your keyboard to move up or down). Go back and change your $2 + 3$ above to $3 + 3$ and re-evaluate it. An empty cell can be *deleted* with backspace.

You can also *edit this text* right here by double clicking on it, which will bring up the TinyMCE Javascript text editor. You can even put embedded mathematics like this $sin(x) - y^3$ by using dollar signs just like in TeX or LaTeX.

Help systems

There are various ways of getting help in Sage.

- navigate through the documentation (there is a link `Help` at the top right of the worksheet),
- `tab` completion,
- contextual help.

We detail below the latter two methods through examples.

Completion and contextual documentation

Start typing something and press the `tab` key. The interface tries to complete it with a command name. If there is more than one completion, then they are all presented to you. Remember that Sage is case sensitive, i.e. it differentiates upper case from lower case. Hence the `tab` completion of `klein` won't show you the `KleinFourGroup` command that builds the group $\mathbf{Z}/2 \times \mathbf{Z}/2$ as a permutation group. Try it on the next cells:

```
sage: klein<tab>

sage: Klein<tab>
```

To see documentation and examples for a command, type a question mark `?` at the end of the command name and press the `tab` key as in:

```
sage: KleinFourGroup?<tab>

sage: # edit here
```

Exercise A

What is the largest prime factor of 600851475143?

```
sage: factor?<tab>

sage: # edit here
```

In the above manipulations we have not stored any data for later use. This can be done in Sage with the = symbol as in:

```
sage: a = 3
sage: b = 2
sage: print a+b
5
```

This can be understood as Sage evaluating the expression to the right of the = sign and creating the appropriate object, and then associating that object with a label, given by the left-hand side (see the foreword of *Tutorial: Objects and Classes in Python and Sage* for details). Multiple assignments can be done at once:

```
sage: a,b = 2,3
sage: print a,b
2 3
```

This allows us to swap the values of two variables directly:

```
sage: a,b = 2,3
sage: a,b = b,a
sage: print a,b
3 2
```

We can also assign a common value to several variables simultaneously:

```
sage: c = d = 1
sage: c, d
(1, 1)
sage: d = 2
sage: c, d
(1, 2)
```

Note that when we use the word *variable* in the computer-science sense we mean "a label attached to some data stored by Sage". Once an object is created, some *methods* apply to it. This means *functions* but instead of writing **f(my_object)** you write **my_object.f**():

```
sage: p = 17
sage: p.is_prime()
True
```

See *Tutorial: Objects and Classes in Python and Sage* for details. To know all methods of an object you can once more use tab-completion. Write the name of the object followed by a dot and then press `tab`:

```
sage: a.<tab>

sage: # edit here
```

Exercise B

Create the permutation 51324 and assign it to the variable `p`.

```
sage: Permutation?<tab>
```

```
sage: # edit here
```

What is the `inverse` of `p`?

```
sage: p.inv<tab>
```

```
sage: # edit here
```

Does `p` have the `pattern` 123? What about 1234? And 312? (even if you don't know what a pattern is, you should be able to find a command that does this).

```
sage: p.pat<tab>
```

```
sage: # edit here
```

Some linear algebra

Exercise C

Use the `matrix()` command to create the following matrix.

$$M = \begin{pmatrix} 10 & 4 & 1 & 1 \\ 4 & 6 & 5 & 1 \\ 1 & 5 & 6 & 4 \\ 1 & 1 & 4 & 10 \end{pmatrix}$$

```
sage: matrix?<tab>
```

```
sage: # edit here
```

Then, using methods of the matrix,

1. Compute the determinant of the matrix.
2. Compute the echelon form of the matrix.
3. Compute the eigenvalues of the matrix.
4. Compute the kernel of the matrix.
5. Compute the LLL decomposition of the matrix (and lookup the documentation for what LLL is if needed!)

```
sage: # edit here
```

```
sage: # edit here
```

Now that you know how to access the different methods of matrices,

6. Create the vector $v = (1, -1, -1, 1)$.
7. Compute the two products: $M \cdot v$ and $v \cdot M$. What mathematically borderline operation is Sage doing implicitly?

```
sage: vector?<tab>
```

```
sage: # edit here
```

Note: Vectors in Sage are row vectors. A method such as `eigenspaces` might not return what you expect, so it is best to specify `eigenspaces_left` or `eigenspaces_right` instead. Same thing for kernel (`left_kernel` or `right_kernel`), and so on.

Some Plotting

The `plot()` command allows you to draw plots of functions. Recall that you can access the documentation by pressing the `tab` key after writing `plot?` in a cell:

```
sage: plot?<tab>
```

```
sage: # edit here
```

Here is a simple example:

```
sage: var('x')   # make sure x is a symbolic variable
x
sage: plot(sin(x^2), (x,0,10))
```

Here is a more complicated plot. Try to change every single input to the plot command in some way, evaluating to see what happens:

```
sage: P = plot(sin(x^2), (x,-2,2), rgbcolor=(0.8,0,0.2), thickness=3, linestyle='--', fill='axis')
sage: show(P, gridlines=True)
```

Above we used the `show()` command to show a plot after it was created. You can also use `P.show` instead:

```
sage: P.show(gridlines=True)
```

Try putting the cursor right after `P.show(` and pressing tab to get a list of the options for how you can change the values of the given inputs.

```
sage: P.show(
```

Plotting multiple functions at once is as easy as adding them together:

```
sage: P1 = plot(sin(x), (x,0,2*pi))
sage: P2 = plot(cos(x), (x,0,2*pi), rgbcolor='red')
sage: P1 + P2
```

Symbolic Expressions

Here is an example of a symbolic function:

```
sage: f(x) = x^4 - 8*x^2 - 3*x + 2
sage: f(x)
x^4 - 8*x^2 - 3*x + 2

sage: f(-3)
20
```

This is an example of a function in the *mathematical* variable x. When Sage starts, it defines the symbol x to be a mathematical variable. If you want to use other symbols for variables, you must define them first:

```
sage: x^2
x^2
sage: u + v
Traceback (most recent call last):
...
NameError: name 'u' is not defined

sage: var('u v')
(u, v)
sage: u + v
u + v
```

Still, it is possible to define symbolic functions without first defining their variables:

```
sage: f(w) = w^2
sage: f(3)
9
```

In this case those variables are defined implicitly:

```
sage: w
w
```

Exercise D

Define the symbolic function $f(x) = x\sin(x^2)$. Plot f on the domain $[-3, 3]$ and color it red. Use the `find_root()` method to numerically approximate the root of f on the interval $[1, 2]$:

```
sage: # edit here
```

Compute the tangent line to f at $x = 1$:

```
sage: # edit here
```

Plot f and the tangent line to f at $x = 1$ in one image:

```
sage: # edit here
```

Exercise E (Advanced)

Solve the following equation for y:

$$y = 1 + xy^2$$

There are two solutions, take the one for which $\lim_{x\to 0} y(x) = 1$. (Don't forget to create the variables x and y!).

```
sage: # edit here
```

Expand y as a truncated Taylor series around 0 and containing $n = 10$ terms.

```
sage: # edit here
```

Do you recognize the coefficients of the Taylor series expansion? You might want to use the On-Line Encyclopedia of Integer Sequences, or better yet, Sage's class `OEIS` which queries the encyclopedia:

```
sage: oeis?<tab>

sage: # edit here
```

Congratulations for completing your first Sage tutorial!

6.1.3 Abelian Sandpile Model

Author: David Perkinson, Reed College

Introduction

These notes provide an introduction to Dhar's abelian sandpile model (ASM) and to Sage Sandpiles, a collection of tools in Sage for doing sandpile calculations. For a more thorough introduction to the theory of the ASM, the papers *Chip-Firing and Rotor-Routing on Directed Graphs* [H], by Holroyd et al. and *Riemann-Roch and Abel-Jacobi Theory on a Finite Graph* by Baker and Norine [BN] are recommended.

To describe the ASM, we start with a *sandpile graph*: a directed multigraph Γ with a vertex s that is accessible from every vertex (except possibly s, itself). By *multigraph*, we mean that each edge of Γ is assigned a nonnegative integer weight. To say s is *accessible* from some vertex v means that there is a sequence of directed edges starting at v and ending at s. We call s the *sink* of the sandpile graph, even though it might have outgoing edges, for reasons that will be made clear in a moment.

We denoted the vertices of Γ by V and define $\tilde{V} = V \setminus \{s\}$.

Configurations and divisors

A *configuration* on Γ is an element of $\mathbb{N}\tilde{V}$, i.e., the assignment of a nonnegative integer to each nonsink vertex. We think of each integer as a number of grains of sand being placed at the corresponding vertex. A *divisor* on Γ is an element of $\mathbb{Z}V$, i.e., an element in the free abelian group on *all* of the vertices. In the context of divisors, it is sometimes useful to think of assigning dollars to each vertex, with negative integers signifying a debt.

Stabilization

A configuration c is *stable* at a vertex $v \in \tilde{V}$ if $c(v) <$ out-degree(v), and c itself is stable if it is stable at each nonsink vertex. Otherwise, c is *unstable*. If c is unstable at v, the vertex v can be *fired* (*toppled*) by removing out-degree(v) grains of sand from v and adding grains of sand to the neighbors of sand, determined by the weights of the edges leaving v.

Despite our best intentions, we sometimes consider firing a stable vertex, resulting in a configuration with a "negative amount" of sand at that vertex. We may also *reverse-firing* a vertex, absorbing sand from the vertex's neighbors.

Example. Consider the graph:

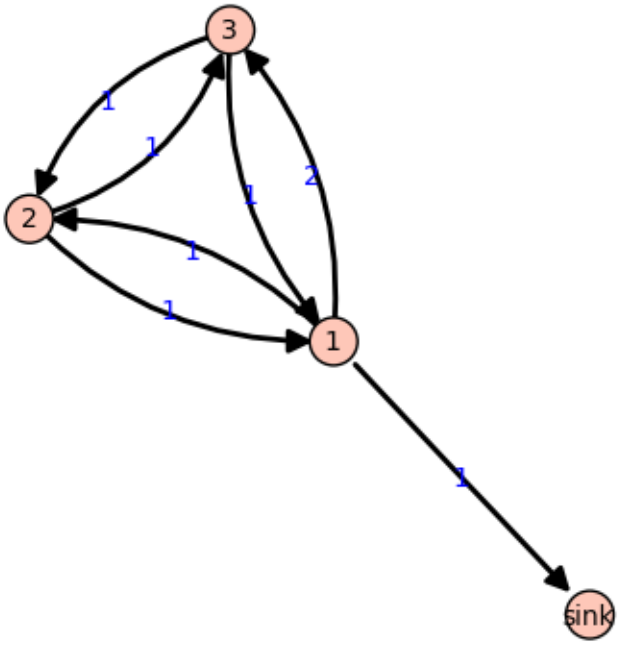


Figure 6.1: Γ

All edges have weight 1 except for the edge from vertex 1 to vertex 3, which has weight 2. If we let $c = (5, 0, 1)$ with the indicated number of grains of sand on vertices 1, 2, and 3, respectively, then only vertex 1, whose out-degree is 4, is unstable. Firing vertex 1 gives a new configuration $c' = (1, 1, 3)$. Here, 4 grains have left vertex 1. One of these has gone to the sink vertex (and forgotten), one has gone to vertex 1, and two have gone to vertex 2, since the edge from 1 to 2 has weight 2. Vertex 3 in the new configuration is now unstable. The Sage code for this example follows.

```
sage: g = {'sink':{},
...        1:{'sink':1, 2:1, 3:2},
...        2:{1:1, 3:1},
...        3:{1:1, 2:1}}
sage: S = Sandpile(g, 'sink')    # create the sandpile
sage: S.show(edge_labels=true)   # display the graph
```

Create the configuration:

```
sage: c = SandpileConfig(S, {1:5, 2:0, 3:1})
sage: S.out_degree()
{1: 4, 2: 2, 3: 2, 'sink': 0}
```

Fire vertex one:

```
sage: c.fire_vertex(1)
{1: 1, 2: 1, 3: 3}
```

The configuration is unchanged:

```
sage: c
{1: 5, 2: 0, 3: 1}
```

Repeatedly fire vertices until the configuration becomes stable:

```
sage: c.stabilize()
{1: 2, 2: 1, 3: 1}
```

Alternatives:

```
sage: ~c              # shorthand for c.stabilize()
{1: 2, 2: 1, 3: 1}
sage: c.stabilize(with_firing_vector=true)
[{1: 2, 2: 1, 3: 1}, {1: 2, 2: 2, 3: 3}]
```

Since vertex 3 has become unstable after firing vertex 1, it can be fired, which causes vertex 2 to become unstable, etc. Repeated firings eventually lead to a stable configuration. The last line of the Sage code, above, is a list, the first element of which is the resulting stable configuration, $(2, 1, 1)$. The second component records how many times each vertex fired in the stabilization.

Since the sink is accessible from each nonsink vertex and never fires, every configuration will stabilize after a finite number of vertex-firings. It is not obvious, but the resulting stabilization is independent of the order in which unstable vertices are fired. Thus, each configuration stabilizes to a unique stable configuration.

Laplacian

Fix an order on the vertices of Γ. The *Laplacian* of Γ is

$$L := D - A$$

where D is the diagonal matrix of out-degrees of the vertices and A is the adjacency matrix whose (i, j)-th entry is the weight of the edge from vertex i to vertex j, which we take to be 0 if there is no edge. The *reduced Laplacian*, $\tilde{L}$, is the submatrix of the Laplacian formed by removing the row and column corresponding to the sink vertex. Firing a vertex of a configuration is the same as subtracting the corresponding row of the reduced Laplacian.

Example. (Continued.)

```
sage: S.vertices()  # the ordering of the vertices
[1, 2, 3, 'sink']
sage: S.laplacian()
[ 4 -1 -2 -1]
[-1  2 -1  0]
[-1 -1  2  0]
[ 0  0  0  0]
sage: S.reduced_laplacian()
[ 4 -1 -2]
[-1  2 -1]
[-1 -1  2]

The configuration we considered previously:

sage: c = SandpileConfig(S, [5,0,1])
sage: c
{1: 5, 2: 0, 3: 1}

Firing vertex 1 is the same as subtracting the
corresponding row from the reduced Laplacian:

sage: c.fire_vertex(1).values()
[1, 1, 3]
sage: S.reduced_laplacian()[0]
(4, -1, -2)
sage: vector([5,0,1]) - vector([4,-1,-2])
(1, 1, 3)
```

Recurrent elements

Imagine an experiment in which grains of sand are dropped one-at-a-time onto a graph, pausing to allow the configuration to stabilize between drops. Some configurations will only be seen once in this process. For example, for most graphs, once sand is dropped on the graph, no addition of sand+stabilization will result in a graph empty of sand. Other configurations—the so-called *recurrent configurations*—will be seen infinitely often as the process is repeated indefinitely.

To be precise, a configuration c is *recurrent* if (i) it is stable, and (ii) given any configuration a, there is a configuration b such that $c = \text{stab}(a + b)$, the stabilization of $a + b$.

The *maximal-stable* configuration, denoted $c_{\max}$ is defined by $c_{\max}(v) = \text{out-degree}(v) - 1$ for all nonsink vertices v. It is clear that $c_{\max}$ is recurrent. Further, it is not hard to see that a configuration is recurrent if and only if it has the form $\text{stab}(a + c_{\max})$ for some configuration a.

Example. (Continued.)

```
sage: S.recurrents(verbose=false)
[[3, 1, 1], [2, 1, 1], [3, 1, 0]]
sage: c = SandpileConfig(S, [2,1,1])
sage: c
{1: 2, 2: 1, 3: 1}
sage: c.is_recurrent()
True
sage: S.max_stable()
{1: 3, 2: 1, 3: 1}

Adding any configuration to the max-stable configuration and stabilizing
yields a recurrent configuration.
```

```
sage: x = SandpileConfig(S, [1,0,0])
sage: x + S.max_stable()
{1: 4, 2: 1, 3: 1}

Use & to add and stabilize:

sage: c = x & S.max_stable()
sage: c
{1: 3, 2: 1, 3: 0}
sage: c.is_recurrent()
True

Note the various ways of performing addition + stabilization:

sage: m = S.max_stable()
sage: (x + m).stabilize() == ~(x + m)
True
sage: (x + m).stabilize() == x & m
True
```

Burning Configuration

A *burning configuration* is a nonnegative integer-linear combination of the rows of the reduced Laplacian matrix having nonnegative entries and such that every vertex has a path from some vertex in its support. The corresponding *burning script* gives the integer-linear combination needed to obtain the burning configuration. So if b is the burning configuration, σ is its script, and $\tilde{L}$ is the reduced Laplacian, then $\sigma\,\tilde{L} = b$. The *minimal burning configuration* is the one with the minimal script (its components are no larger than the components of any other script for a burning configuration).

The following are equivalent for a configuration c with burning configuration b having script σ:

- c is recurrent;
- $c + b$ stabilizes to c;
- the firing vector for the stabilization of $c + b$ is σ.

The burning configuration and script are computed using a modified version of Speer's script algorithm. This is a generalization to directed multigraphs of Dhar's burning algorithm.

Example.

```
sage: g = {0:{},1:{0:1,3:1,4:1},2:{0:1,3:1,5:1},
...        3:{2:1,5:1},4:{1:1,3:1},5:{2:1,3:1}}
sage: G = Sandpile(g,0)
sage: G.burning_config()
{1: 2, 2: 0, 3: 1, 4: 1, 5: 0}
sage: G.burning_config().values()
[2, 0, 1, 1, 0]
sage: G.burning_script()
{1: 1, 2: 3, 3: 5, 4: 1, 5: 4}
sage: G.burning_script().values()
[1, 3, 5, 1, 4]
sage: matrix(G.burning_script().values())*G.reduced_laplacian()
[2 0 1 1 0]
```

Sandpile group

The collection of stable configurations forms a commutative monoid with addition defined as ordinary addition followed by stabilization. The identity element is the all-zero configuration. This monoid is a group exactly when the underlying graph is a DAG (directed acyclic graph).

The recurrent elements form a submonoid which turns out to be a group. This group is called the *sandpile group* for Γ, denoted $\mathcal{S}(\Gamma)$. Its identity element is usually not the all-zero configuration (again, only in the case that Γ is a DAG). So finding the identity element is an interesting problem.

Let $n = |V| - 1$ and fix an ordering of the nonsink vertices. Let $\tilde{\mathcal{L}} \subset \mathbb{Z}^n$ denote the column-span of $\tilde{L}^t$, the transpose of the reduced Laplacian. It is a theorem that

$$\mathcal{S}(\Gamma) \approx \mathbb{Z}^n / \tilde{\mathcal{L}}.$$

Thus, the number of elements of the sandpile group is $\det \tilde{L}$, which by the matrix-tree theorem is the number of weighted trees directed into the sink.

Example. (Continued.)

```
sage: S.group_order()
3
sage: S.invariant_factors()
[1, 1, 3]
sage: S.reduced_laplacian().dense_matrix().smith_form()
(
[1 0 0]  [ 0  0  1]  [3 1 4]
[0 1 0]  [ 1  0  0]  [4 1 6]
[0 0 3], [ 0  1 -1], [4 1 5]
)

Adding the identity to any recurrent configuration and stabilizing yields
the same recurrent configuration:

sage: S.identity()
{1: 3, 2: 1, 3: 0}
sage: i = S.identity()
sage: m = S.max_stable()
sage: i & m == m
True
```

Self-organized criticality

The sandpile model was introduced by Bak, Tang, and Wiesenfeld in the paper, *Self-organized criticality: an explanation of 1/f noise* [BTW]. The term *self-organized criticality* has no precise definition, but can be loosely taken to describe a system that naturally evolves to a state that is barely stable and such that the instabilities are described by a power law. In practice, *self-organized criticality* is often taken to mean *like the sandpile model on a grid-graph*. The grid graph is just a grid with an extra sink vertex. The vertices on the interior of each side have one edge to the sink, and the corner vertices have an edge of weight 2. Thus, every nonsink vertex has out-degree 4.

Imagine repeatedly dropping grains of sand on and empty grid graph, allowing the sandpile to stabilize in between. At first there is little activity, but as time goes on, the size and extent of the avalanche caused by a single grain of sand becomes hard to predict. Computer experiments—I do not think there is a proof, yet—indicate that the distribution of avalanche sizes obeys a power law with exponent -1. In the example below, the size of an avalanche is taken to be the sum of the number of times each vertex fires.

Example (distribution of avalanche sizes).

```
sage: S = grid_sandpile(10,10)
sage: m = S.max_stable()
sage: a = []
sage: for i in range(10000):  # long time (15s on sage.math, 2012)
...       m = m.add_random()
...       m, f = m.stabilize(true)
...       a.append(sum(f.values()))
...
sage: p = list_plot([[log(i+1),log(a.count(i))] for i in [0..max(a)] if a.count(i)])  # long time
sage: p.axes_labels(['log(N)','log(D(N))'])  # long time
sage: p  # long time
```

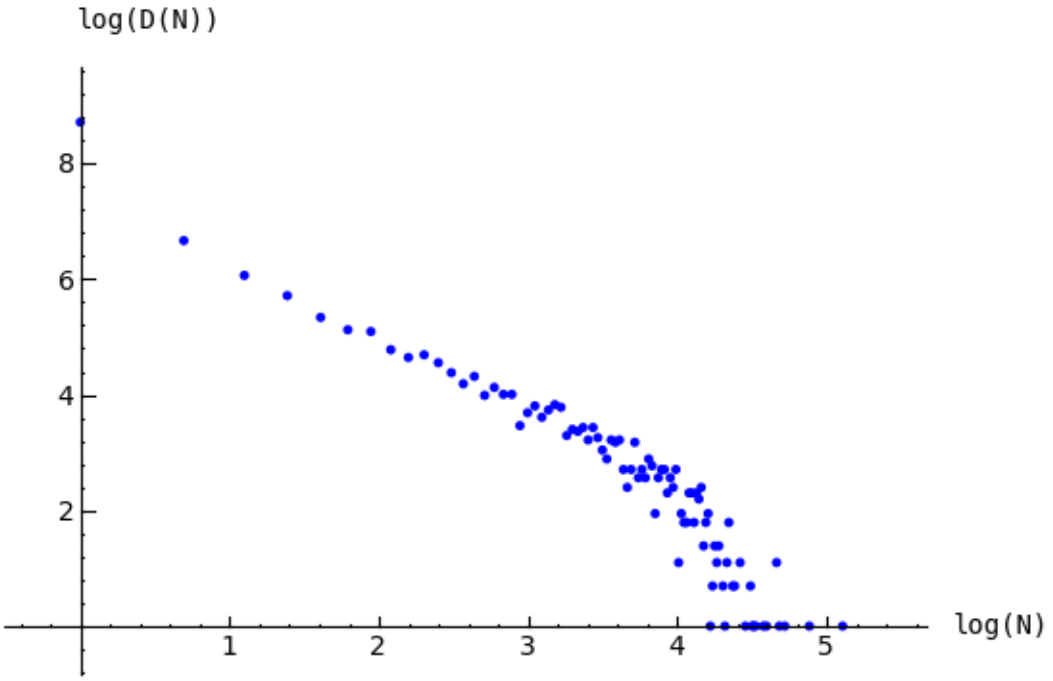


Figure 6.2: Distribution of avalanche sizes

Note: In the above code, `m.stabilize(true)` returns a list consisting of the stabilized configuration and the firing vector. (Omitting `true` would give just the stabilized configuration.)

Divisors and Discrete Riemann surfaces

A reference for this section is *Riemann-Roch and Abel-Jacobi theory on a finite graph* [BN].

A *divisor* on Γ is an element of the free abelian group on its vertices, including the sink. Suppose, as above, that the $n+1$ vertices of Γ have been ordered, and that $\mathcal{L}$ is the column span of the transpose of the Laplacian. A divisor is then identified with an element $D \in \mathbb{Z}^{n+1}$ and two divisors are *linearly equivalent* if they differ by an element of $\mathcal{L}$. A divisor E is *effective*, written $E \geq 0$, if $E(v) \geq 0$ for each $v \in V$, i.e., if $E \in \mathbb{N}^{n+1}$. The *degree* of a divisor, D, is $deg(D) := \sum_{v \in V} D(v)$. The divisors of degree zero modulo linear equivalence form the *Picard group*, or *Jacobian* of the graph. For an undirected graph, the Picard group is isomorphic to the sandpile group.

The *complete linear system* for a divisor D, denoted $|D|$, is the collection of effective divisors linearly equivalent to D.

Riemann-Roch To describe the Riemann-Roch theorem in this context, suppose that Γ is an undirected, unweighted graph. The *dimension*, $r(D)$ of the linear system $|D|$ is -1 if $|D| = \emptyset$ and otherwise is the greatest integer s such that $|D - E| \neq 0$ for all effective divisors E of degree s. Define the *canonical divisor* by $K = \sum_{v \in V} (\deg(v) - 2)v$ and the *genus* by $g = \#(E) - \#(V) + 1$. The Riemann-Roch theorem says that for any divisor D,

$$r(D) - r(K - D) = \deg(D) + 1 - g.$$

Example. (Some of the following calculations require the installation of *4ti2*.)

```
sage: G = complete_sandpile(5)  # the sandpile on the complete graph with 5 vertices

The genus (num_edges method counts each undirected edge twice):

sage: g = G.num_edges()/2 - G.num_verts() + 1

A divisor on the graph:

sage: D = SandpileDivisor(G, [1,2,2,0,2])

Verify the Riemann-Roch theorem:

sage: K = G.canonical_divisor()
sage: D.r_of_D() - (K - D).r_of_D() == D.deg() + 1 - g # optional - 4ti2
True

The effective divisors linearly equivalent to D:

sage: [E.values() for E in D.effective_div()]  # optional - 4ti2
[[0, 1, 1, 4, 1], [4, 0, 0, 3, 0], [1, 2, 2, 0, 2]]

The nonspecial divisors up to linear equivalence (divisors of degree
g-1 with empty linear systems)

sage: N = G.nonspecial_divisors()
sage: [E.values() for E in N[:5]]   # the first few
[[-1, 2, 1, 3, 0],
 [-1, 0, 3, 1, 2],
 [-1, 2, 0, 3, 1],
 [-1, 3, 1, 2, 0],
 [-1, 2, 0, 1, 3]]
sage: len(N)
24
sage: len(N) == G.h_vector()[-1]
True
```

Picturing linear systems Fix a divisor D. There are at least two natural graphs associated with linear system associated with D. First, consider the directed graph with vertex set $|D|$ and with an edge from vertex E to vertex F if F is attained from E by firing a single unstable vertex.

```
sage: S = Sandpile(graphs.CycleGraph(6),0)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: D.is_alive()
True
sage: eff = D.effective_div() # optional - 4ti2
sage: firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01,iterations=500) # optional - 4ti2
```

The second graph has the same set of vertices but with an edge from E to F if F is obtained from E by firing all unstable vertices of E.

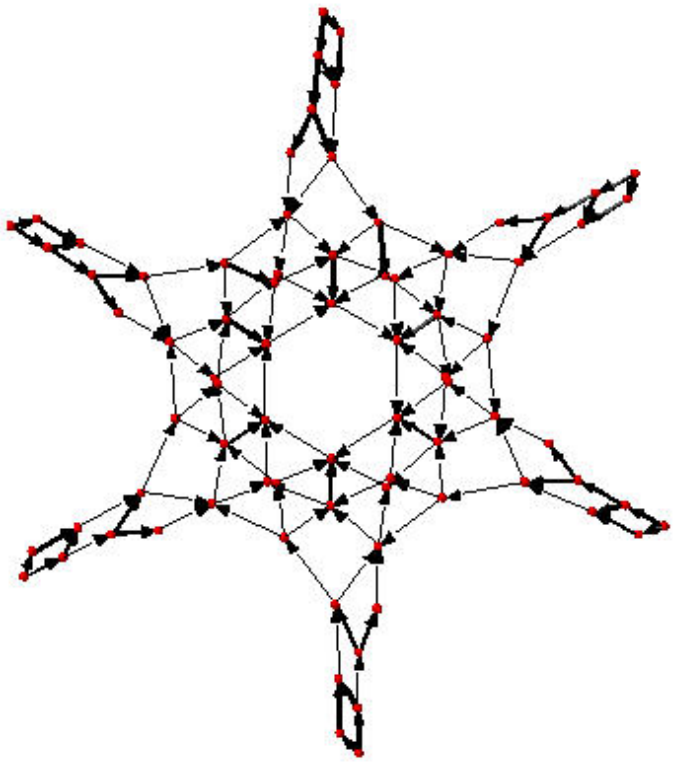


Figure 6.3: Complete linear system for (1,1,1,1,2,0) on C_6: single firings

```
sage: S = Sandpile(graphs.CycleGraph(6),0)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: eff = D.effective_div()  # optional - 4ti2
sage: parallel_firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01,iterations=500) # optional
```

Note that in each of the examples, above, starting at any divisor in the linear system and following edges, one is eventually led into a cycle of length 6 (cycling the divisor (1,1,1,1,2,0)). Thus, `D.alive()` returns `True`. In Sage, one would be able to rotate the above figures to get a better idea of the structure.

Algebraic geometry of sandpiles

Affine Let $n = |V| - 1$, and fix an ordering on the nonsink vertices of Γ. let $\tilde{\mathcal{L}} \subset \mathbb{Z}^n$ denote the column-span of $\tilde{L}^t$, the transpose of the reduced Laplacian. Label vertex i with the indeterminate x_i, and let $\mathbb{C}[\Gamma_s] = \mathbb{C}[x_1, \ldots, x_n]$. (Here, s denotes the sink vertex of Γ.) The *sandpile ideal* or *toppling ideal*, first studied by Cori, Rossin, and Salvy [CRS] for undirected graphs, is the lattice ideal for $\tilde{\mathcal{L}}$:

$$I = I(\Gamma_s) := \{x^u - x^v : u - v \in \tilde{\mathcal{L}}\} \subset \mathbb{C}[\Gamma_s],$$

where $x^u := \prod_{i=1}^n x^{u_i}$ for $u \in \mathbb{Z}^n$.

For each $c \in \mathbb{Z}^n$ define $t(c) = x^{c^+} - x^{c^-}$ where $c_i^+ = \max\{c_i, 0\}$ and $c^- = \max\{-c_i, 0\}$ so that $c = c^+ - c^-$. Then, for each $\sigma \in \mathbb{Z}^n$, define $T(\sigma) = t(\tilde{L}^t \sigma)$. It then turns out that

$$I = (T(e_1), \ldots, T(e_n), x^b - 1)$$

where e_i is the i-th standard basis vector and b is any burning configuration.

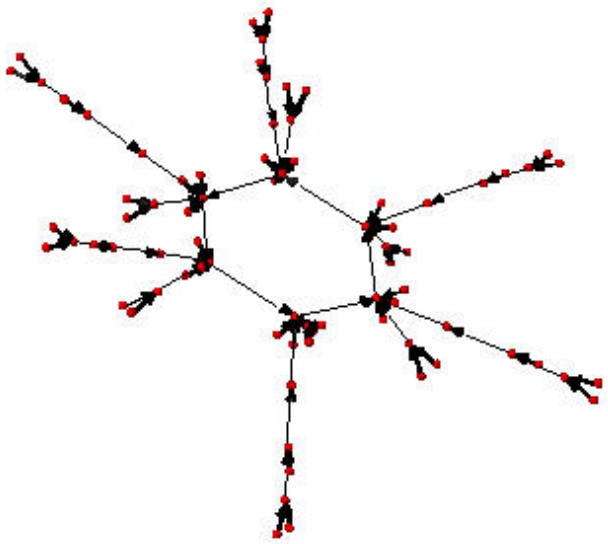


Figure 6.4: Complete linear system for (1,1,1,1,2,0) on C_6: parallel firings

The affine coordinate ring, $\mathbb{C}[\Gamma_s]/I$, is isomorphic to the group algebra of the sandpile group, $\mathbb{C}[\mathcal{S}(\Gamma)]$.

The standard term-ordering on $\mathbb{C}[\Gamma_s]$ is graded reverse lexigraphical order with $x_i > x_j$ if vertex v_i is further from the sink than vertex v_j. (There are choices to be made for vertices equidistant from the sink). If σ_b is the script for a burning configuration (not necessarily minimal), then

$$\{T(\sigma) : \sigma \leq \sigma_b\}$$

is a Groebner basis for I.

Projective Now let $\mathbb{C}[\Gamma] = \mathbb{C}[x_0, x_1, \ldots, x_n]$, where x_0 corresponds to the sink vertex. The *homogeneous sandpile ideal*, denoted I^h, is obtaining by homogenizing I with respect to x_0. Let L be the (full) Laplacian, and $\mathcal{L} \subset \mathbb{Z}^{n+1}$ be the column span of its transpose, L^t. Then I^h is the lattice ideal for $\mathcal{L}$:

$$I^h = I^h(\Gamma) := \{x^u - x^v : u - v \in \mathcal{L}\} \subset \mathbb{C}[\Gamma].$$

This ideal can be calculated by saturating the ideal

$$(T(e_i) : i = 0, \ldots n)$$

with respect to the product of the indeterminates: $\prod_{i=0}^{n} x_i$ (extending the T operator in the obvious way). A Groebner basis with respect to the degree lexicographic order describe above (with x_0 the smallest vertex), is obtained by homogenizing each element of the Groebner basis for the non-homogeneous sandpile ideal with respect to x_0.

Example.

```
sage: g = {0:{},1:{0:1,3:1,4:1},2:{0:1,3:1,5:1},
...        3:{2:1,5:1},4:{1:1,3:1},5:{2:1,3:1}}
sage: S = Sandpile(g, 0)
sage: S.ring()
Multivariate Polynomial Ring in x5, x4, x3, x2, x1, x0 over Rational Field
```

The homogeneous sandpile ideal:

```
sage: S.ideal()
Ideal (x2 - x0, x3^2 - x5*x0, x5*x3 - x0^2, x4^2 - x3*x1, x5^2 - x3*x0,
x1^3 - x4*x3*x0, x4*x1^2 - x5*x0^2) of Multivariate Polynomial Ring
in x5, x4, x3, x2, x1, x0 over Rational Field
```

The generators of the ideal:

```
sage: S.ideal(true)
[x2 - x0,
 x3^2 - x5*x0,
 x5*x3 - x0^2,
 x4^2 - x3*x1,
 x5^2 - x3*x0,
 x1^3 - x4*x3*x0,
 x4*x1^2 - x5*x0^2]
```

Its resolution:

```
sage: S.resolution() # long time
'R^1 <-- R^7 <-- R^19 <-- R^25 <-- R^16 <-- R^4'
```

and Betti table:

```
sage: S.betti() # long time
            0     1     2     3     4     5
------------------------------------------
    0:      1     1     -     -     -     -
    1:      -     4     6     2     -     -
    2:      -     2     7     7     2     -
    3:      -     -     6    16    14     4
------------------------------------------
total:      1     7    19    25    16     4
```

The Hilbert function:

```
sage: S.hilbert_function()
[1, 5, 11, 15]
```

and its first differences (which counts the number of superstable configurations in each degree):

```
sage: S.h_vector()
[1, 4, 6, 4]
sage: x = [i.deg() for i in S.superstables()]
sage: sorted(x)
[0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3]
```

The degree in which the Hilbert function equals the Hilbert polynomial, the latter always being a constant in the case of a sandpile ideal:

```
sage: S.postulation()
3
```

Zeros The *zero set* for the sandpile ideal I is

$$Z(I) = \{p \in \mathbb{C}^n : f(p) = 0 \text{ for all } f \in I\},$$

the set of simultaneous zeros of the polynomials in I. Letting S^1 denote the unit circle in the complex plane, $Z(I)$ is a finite subgroup of $S^1 \times \cdots \times S^1 \subset \mathbb{C}^n$, isomorphic to the sandpile group. The zero set is actually linearly isomorphic to a faithful representation of the sandpile group on $\mathbb{C}^n$.

Example. (Continued.)

```
sage: S = Sandpile({0: {}, 1: {2: 2}, 2: {0: 4, 1: 1}}, 0)
sage: S.ideal().gens()
[x1^2 - x2^2, x1*x2^3 - x0^4, x2^5 - x1*x0^4]

Approximation to the zero set (setting ``x_0 = 1``):

sage: S.solve()
[[-0.707107 + 0.707107*I, 0.707107 - 0.707107*I],
[-0.707107 - 0.707107*I, 0.707107 + 0.707107*I],
[-I, -I],
[I, I],
[0.707107 + 0.707107*I, -0.707107 - 0.707107*I],
[0.707107 - 0.707107*I, -0.707107 + 0.707107*I],
[1, 1],
[-1, -1]]
sage: len(_) == S.group_order()
True

The zeros are generated as a group by a single vector:

sage: S.points()
[[e^(1/4*I*pi), e^(-3/4*I*pi)]]
```

Resolutions The homogeneous sandpile ideal, I^h, has a free resolution graded by the divisors on Γ modulo linear equivalence. (See the section on *Discrete Riemann Surfaces* for the language of divisors and linear equivalence.) Let $S = \mathbb{C}[\Gamma] = \mathbb{C}[x_0, \ldots, x_n]$, as above, and let $\mathfrak{S}$ denote the group of divisors modulo rational equivalence. Then S is graded by $\mathfrak{S}$ by letting $\deg(x^c) = c \in \mathfrak{S}$ for each monomial x^c. The minimal free resolution of I^h has the form

$$0 \leftarrow I^h \leftarrow \oplus_{D\in\mathfrak{S}} S(-D)^{\beta_{0,D}} \leftarrow \oplus_{D\in\mathfrak{S}} S(-D)^{\beta_{1,D}} \leftarrow \cdots \leftarrow \oplus_{D\in\mathfrak{S}} S(-D)^{\beta_{r,D}} \leftarrow 0.$$

where the $\beta_{i,D}$ are the *Betti numbers* for I^h.

For each divisor class $D \in \mathfrak{S}$, define a simplicial complex,

$$\Delta_D := \{I \subseteq \{0, \ldots, n\} : I \subseteq \text{supp}(E) \text{ for some } E \in |D|\}.$$

The Betti number $\beta_{i,D}$ equals the dimension over $\mathbb{C}$ of the i-th reduced homology group of Δ_D:

$$\beta_{i,D} = \dim_{\mathbb{C}} \tilde{H}_i(\Delta_D; \mathbb{C}).$$

```
sage: S = Sandpile({0:{},1:{0: 1, 2: 1, 3: 4},2:{3: 5},3:{1: 1, 2: 1}},0)

Representatives of all divisor classes with nontrivial homology:

sage: p = S.betti_complexes() # optional - 4ti2
sage: p[0] # optional - 4ti2
[{0: -8, 1: 5, 2: 4, 3: 1},
 Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2), (3,)}]

The homology associated with the first divisor in the list:

sage: D = p[0][0]  # optional - 4ti2
sage: D.effective_div() # optional - 4ti2
[{0: 0, 1: 1, 2: 1, 3: 0}, {0: 0, 1: 0, 2: 0, 3: 2}]
sage: [E.support() for E in D.effective_div()] # optional - 4ti2
[[1, 2], [3]]
sage: D.Dcomplex() # optional - 4ti2
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2), (3,)}
sage: D.Dcomplex().homology() # optional - 4ti2
{0: Z, 1: 0}


The minimal free resolution:

sage: S.resolution()
'R^1 <-- R^5 <-- R^5 <-- R^1'
sage: S.betti()
            0     1     2     3
------------------------------
    0:      1     -     -     -
    1:      -     5     5     -
    2:      -     -     -     1
------------------------------
total:      1     5     5     1
sage: len(p) # optional - 4ti2
11

The degrees and ranks of the homology groups for each element of the list p
(compare with the Betti table, above):

sage: [[sum(d[0].values()),d[1].betti()] for d in p] # optional - 4ti2
[[2, {0: 2, 1: 0}],
[3, {0: 1, 1: 1, 2: 0}],
[2, {0: 2, 1: 0}],
[3, {0: 1, 1: 1, 2: 0}],
[2, {0: 2, 1: 0}],
[3, {0: 1, 1: 1, 2: 0}],
[2, {0: 2, 1: 0}],
[3, {0: 1, 1: 1}],
[2, {0: 2, 1: 0}],
[3, {0: 1, 1: 1, 2: 0}],
[5, {0: 1, 1: 0, 2: 1}]]
```

Complete Intersections and Arithmetically Gorenstein toppling ideals NOTE: in the previous section note that the resolution always has length n since the ideal is Cohen-Macaulay.

To do.

Betti numbers for undirected graphs To do.

4ti2 installation

Warning: The methods for computing linear systems of divisors and their corresponding simplicial complexes require the installation of 4ti2.

To install 4ti2:

Go to the Sage website and look for the *precise name* of the 4ti2 package and install it according to the instructions given there. For instance, suppose the package is named 4ti2.p0.spkg. Install the package with the following command from a UNIX shell prompt:

```
sage -i 4ti2.p0
```

Usage

Initialization

There are three main classes for sandpile structures in Sage: `Sandpile`, `SandpileConfig`, and `SandpileDivisor`. Initialization for `Sandpile` has the form

```
sage: S = Sandpile(graph, sink)
```

where `graph` represents a graph and `sink` is the key for the sink vertex. There are four possible forms for `graph`:

1. a Python dictionary of dictionaries:

```
sage: g = {0: {}, 1: {0: 1, 3: 1, 4: 1}, 2: {0: 1, 3: 1, 5: 1},
...        3: {2: 1, 5: 1}, 4: {1: 1, 3: 1}, 5: {2: 1, 3: 1}}
```

Each key is the name of a vertex. Next to each vertex name v is a dictionary consisting of pairs: `vertex: weight`. Each pair represents a directed edge emanating from v and ending at `vertex` having (non-negative integer) weight equal to `weight`. Loops are allowed. In the example above, all of the weights are 1.

2. a Python dictionary of lists:

```
sage: g = {0: [], 1: [0, 3, 4], 2: [0, 3, 5],
...        3: [2, 5], 4: [1, 3], 5: [2, 3]}
```

This is a short-hand when all of the edge-weights are equal to 1. The above example is for the same displayed graph.

3. a Sage graph (of type `sage.graphs.graph.Graph`):

```
sage: g = graphs.CycleGraph(5)
sage: S = Sandpile(g, 0)
sage: type(g)
<class 'sage.graphs.graph.Graph'>
```

To see the types of built-in graphs, type `graphs.`, including the period, and hit TAB.

4. a Sage digraph:

```
sage: S = Sandpile(digraphs.RandomDirectedGNC(6), 0)
sage: S.show()
```

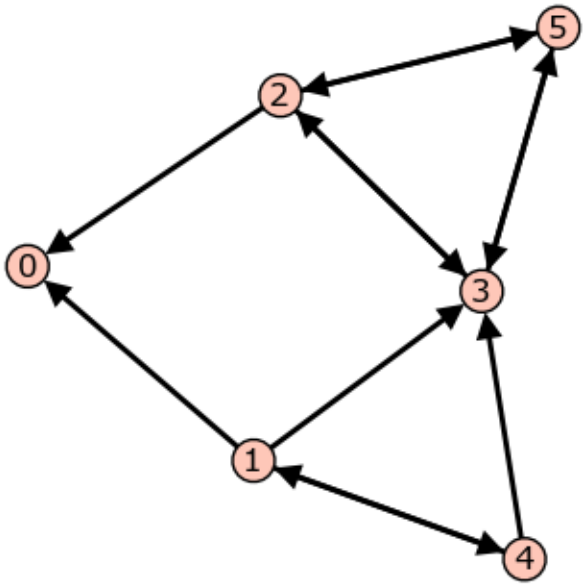


Figure 6.5: Graph from dictionary of dictionaries.

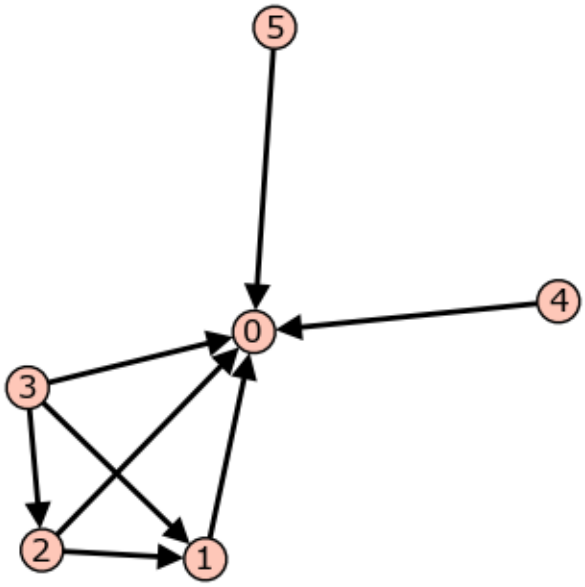


Figure 6.6: A random graph.

See sage.graphs.graph_generators for more information on the Sage graph library and graph constructors.

Each of these four formats is preprocessed by the Sandpile class so that, internally, the graph is represented by the dictionary of dictionaries format first presented. This internal format is returned by `dict()`:

```
sage: S = Sandpile({0:[], 1:[0, 3, 4], 2:[0, 3, 5], 3: [2, 5], 4: [1, 3], 5: [2, 3]},0)
sage: S.dict()
{0: {},
 1: {0: 1, 3: 1, 4: 1},
 2: {0: 1, 3: 1, 5: 1},
 3: {2: 1, 5: 1},
 4: {1: 1, 3: 1},
 5: {2: 1, 3: 1}}
```

Note: The user is responsible for assuring that each vertex has a directed path into the designated sink. If the sink has out-edges, these will be ignored for the purposes of sandpile calculations (but not calculations on divisors).

Code for checking whether a given vertex is a sink:

```
sage: S = Sandpile({0:[], 1:[0, 3, 4], 2:[0, 3, 5], 3: [2, 5], 4: [1, 3], 5: [2, 3]},0)
sage: [S.distance(v,0) for v in S.vertices()] # 0 is a sink
[0, 1, 1, 2, 2, 2]
sage: [S.distance(v,1) for v in S.vertices()] # 1 is not a sink
[+Infinity, 0, +Infinity, +Infinity, 1, +Infinity]
```

Methods

Here are summaries of `Sandpile`, `SandpileConfig`, and `SandpileDivisor` methods (functions). Each summary is followed by a list of complete descriptions of the methods. There are many more methods available for a Sandpile, e.g., those inherited from the class DiGraph. To see them all, enter `dir(Sandpile)` or type `Sandpile.`, including the period, and hit TAB.

Sandpile Summary of methods.

- *all_k_config(k)* — The configuration with all values set to k.
- *all_k_div(k)* — The divisor with all values set to k.
- *betti(verbose=True)* — The Betti table for the homogeneous sandpile ideal.
- *betti_complexes()* — The divisors with nonempty linear systems along with their simplicial complexes.
- *burning_config()* — A minimal burning configuration.
- *burning_script()* — A script for the minimal burning configuration.
- *canonical_divisor()* — The canonical divisor (for undirected graphs).
- *dict()* — A dictionary of dictionaries representing a directed graph.
- *groebner()* — Groebner basis for the homogeneous sandpile ideal with respect to the standard sandpile ordering.
- *group_order()* — The size of the sandpile group.
- *h_vector()* — The first differences of the Hilbert function of the homogeneous sandpile ideal.
- *hilbert_function()* — The Hilbert function of the homogeneous sandpile ideal.
- *ideal(gens=False)* — The saturated, homogeneous sandpile ideal.

- *identity()* — The identity configuration.
- *in_degree(v=None)* — The in-degree of a vertex or a list of all in-degrees.
- *invariant_factors()* — The invariant factors of the sandpile group (a finite abelian group).
- *is_undirected()* — `True` if `(u,v)` is and edge if and only if `(v,u)` is an edges, each edge with the same weight.
- *laplacian()* — The Laplacian matrix of the graph.
- *max_stable()* — The maximal stable configuration.
- *max_stable_div()* — The maximal stable divisor.
- *max_superstables(verbose=True)* — The maximal superstable configurations.
- *min_recurrents(verbose=True)* — The minimal recurrent elements.
- *nonsink_vertices()* — The names of the nonsink vertices.
- *nonspecial_divisors(verbose=True)* — The nonspecial divisors (only for undirected graphs).
- *num_edges()* — The number of edges.
- *num_verts()* — The number of vertices.
- *out_degree(v=None)* — The out-degree of a vertex or a list of all out-degrees.
- *points()* — Generators for the multiplicative group of zeros of the sandpile ideal.
- *postulation()* — The postulation number of the sandpile ideal.
- *recurrents(verbose=True)* — The list of recurrent configurations.
- *reduced_laplacian()* — The reduced Laplacian matrix of the graph.
- *reorder_vertices()* — Create a copy of the sandpile but with the vertices reordered.
- *resolution(verbose=False)* — The minimal free resolution of the homogeneous sandpile ideal.
- *ring()* — The ring containing the homogeneous sandpile ideal.
- *show(kwds)* — Draws the graph.
- *show3d(kwds)* — Draws the graph.
- *sink()* — The identifier for the sink vertex.
- *solve()* — Approximations of the complex affine zeros of the sandpile ideal.
- *superstables(verbose=True)* — The list of superstable configurations.
- *symmetric_recurrents(orbits)* — The list of symmetric recurrent configurations.
- *unsaturated_ideal()* — The unsaturated, homogeneous sandpile ideal.
- *version()* — The version number of Sage Sandpiles.
- *vertices(key=None, boundary_first=False)* — List of the vertices.
- *zero_config()* — The all-zero configuration.
- *zero_div()* — The all-zero divisor.

Complete descriptions of Sandpile methods.

— **all_k_config(k)**

The configuration with all values set to k.

INPUT:

`k` - integer

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.all_k_config(7)
{1: 7, 2: 7, 3: 7, 4: 7, 5: 7}
```

— **all_k_div(k)**

The divisor with all values set to k.

INPUT:

`k` - integer

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.all_k_div(7)
{0: 7, 1: 7, 2: 7, 3: 7, 4: 7, 5: 7}
```

— **betti(verbose=True)**

Computes the Betti table for the homogeneous sandpile ideal. If `verbose` is `True`, it prints the standard Betti table, otherwise, it returns a less formated table.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

Betti numbers for the sandpile

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.betti() # long time
            0     1     2     3     4     5
-------------------------------------------
    0:      1     1     -     -     -     -
    1:      -     4     6     2     -     -
    2:      -     2     7     7     2     -
    3:      -     -     6    16    14     4
-------------------------------------------
total:      1     7    19    25    16     4
sage: S.betti(False) # long time
[1, 7, 19, 25, 16, 4]
```

— **betti_complexes()**

A list of all the divisors with nonempty linear systems whose corresponding simplicial complexes have nonzero homology in some dimension. Each such divisor is returned with its corresponding simplicial complex.

INPUT:

None

OUTPUT:

list (of pairs [divisors, corresponding simplicial complex])

EXAMPLES:

```
sage: S = Sandpile({0:{},1:{0: 1, 2: 1, 3: 4},2:{3: 5},3:{1: 1, 2: 1}},0)
sage: p = S.betti_complexes() # optional - 4ti2
sage: p[0] # optional - 4ti2
[{0: -8, 1: 5, 2: 4, 3: 1},
 Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2), (3,)}]
sage: S.resolution()
'R^1 <-- R^5 <-- R^5 <-- R^1'
sage: S.betti()
            0     1     2     3
------------------------------
    0:      1     -     -     -
    1:      -     5     5     -
    2:      -     -     -     1
------------------------------
total:      1     5     5     1
sage: len(p) # optional - 4ti2
11
sage: p[0][1].homology() # optional - 4ti2
{0: Z, 1: 0}
sage: p[-1][1].homology() # optional - 4ti2
{0: 0, 1: 0, 2: Z}
```

— **burning_config()**

A minimal burning configuration.

INPUT:

None

OUTPUT:

dict (configuration)

EXAMPLES:

```
sage: g = {0:{},1:{0:1,3:1,4:1},2:{0:1,3:1,5:1},\
           3:{2:1,5:1},4:{1:1,3:1},5:{2:1,3:1}}
sage: S = Sandpile(g,0)
sage: S.burning_config()
{1: 2, 2: 0, 3: 1, 4: 1, 5: 0}
sage: S.burning_config().values()
[2, 0, 1, 1, 0]
sage: S.burning_script()
{1: 1, 2: 3, 3: 5, 4: 1, 5: 4}
sage: script = S.burning_script().values()
sage: script
[1, 3, 5, 1, 4]
```

```
sage: matrix(script)*S.reduced_laplacian()
[2 0 1 1 0]
```

NOTES:

The burning configuration and script are computed using a modified version of Speer's script algorithm. This is a generalization to directed multigraphs of Dhar's burning algorithm.

A *burning configuration* is a nonnegative integer-linear combination of the rows of the reduced Laplacian matrix having nonnegative entries and such that every vertex has a path from some vertex in its support. The corresponding *burning script* gives the integer-linear combination needed to obtain the burning configuration. So if b is the burning configuration, sigma is its script, and tilde{L} is the reduced Laplacian, then sigma * tilde{L} = b. The *minimal burning configuration* is the one with the minimal script (its components are no larger than the components of any other script for a burning configuration).

The following are equivalent for a configuration c with burning configuration b having script sigma:

- c is recurrent;
- c+b stabilizes to c;
- the firing vector for the stabilization of c+b is sigma.

— **burning_script**()

A script for the minimal burning configuration.

INPUT:

None

OUTPUT:

dict

EXAMPLES:

```
sage: g = {0:{},1:{0:1,3:1,4:1},2:{0:1,3:1,5:1},\
3:{2:1,5:1},4:{1:1,3:1},5:{2:1,3:1}}
sage: S = Sandpile(g,0)
sage: S.burning_config()
{1: 2, 2: 0, 3: 1, 4: 1, 5: 0}
sage: S.burning_config().values()
[2, 0, 1, 1, 0]
sage: S.burning_script()
{1: 1, 2: 3, 3: 5, 4: 1, 5: 4}
sage: script = S.burning_script().values()
sage: script
[1, 3, 5, 1, 4]
sage: matrix(script)*S.reduced_laplacian()
[2 0 1 1 0]
```

NOTES:

The burning configuration and script are computed using a modified version of Speer's script algorithm. This is a generalization to directed multigraphs of Dhar's burning algorithm.

A *burning configuration* is a nonnegative integer-linear combination of the rows of the reduced Laplacian matrix having nonnegative entries and such that every vertex has a path from some vertex in its support. The corresponding *burning script* gives the integer-linear combination needed to obtain the burning configuration. So if b is the burning configuration, s is its script, and L_{mathrm{red}} is the reduced Laplacian, then s * L_{mathrm{red}}= b. The *minimal burning configuration* is the one with the minimal script (its components are no larger than the components of any other script for a burning configuration).

The following are equivalent for a configuration c with burning configuration b having script s:

- c is recurrent;
- c+b stabilizes to c;
- the firing vector for the stabilization of c+b is s.

— **canonical_divisor()**

The canonical divisor: the divisor `deg(v)-2` grains of sand on each vertex. Only for undirected graphs.

INPUT:

None

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = complete_sandpile(4)
sage: S.canonical_divisor()
{0: 1, 1: 1, 2: 1, 3: 1}
```

— **dict()**

A dictionary of dictionaries representing a directed graph.

INPUT:

None

OUTPUT:

dict

EXAMPLES:

```
sage: G = sandlib('generic')
sage: G.dict()
{0: {},
 1: {0: 1, 3: 1, 4: 1},
 2: {0: 1, 3: 1, 5: 1},
 3: {2: 1, 5: 1},
 4: {1: 1, 3: 1},
 5: {2: 1, 3: 1}}
sage: G.sink()
0
```

— **groebner()**

A Groebner basis for the homogeneous sandpile ideal with respect to the standard sandpile ordering (see `ring`).

INPUT:

None

OUTPUT:

Groebner basis

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.groebner()
[x4*x1^2 - x5*x0^2, x1^3 - x4*x3*x0, x5^2 - x3*x0, x4^2 - x3*x1, x5*x3 - x0^2,
x3^2 - x5*x0, x2 - x0]
```

— **group_order()**

The size of the sandpile group.

INPUT:

None

OUTPUT:

int

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.group_order()
15
```

— **h_vector()**

The first differences of the Hilbert function of the homogeneous sandpile ideal. It lists the number of superstable configurations in each degree.

INPUT:

None

OUTPUT:

list of nonnegative integers

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.hilbert_function()
[1, 5, 11, 15]
sage: S.h_vector()
[1, 4, 6, 4]
```

— **hilbert_function()**

The Hilbert function of the homogeneous sandpile ideal.

INPUT:

None

OUTPUT:

list of nonnegative integers

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.hilbert_function()
[1, 5, 11, 15]
```

— **ideal(gens=False)**

The saturated, homogeneous sandpile ideal (or its generators if `gens=True`).

INPUT:

`verbose` (optional) - boolean

OUTPUT:

ideal or, optionally, the generators of an ideal

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.ideal()
Ideal (x2 - x0, x3^2 - x5*x0, x5*x3 - x0^2, x4^2 - x3*x1, x5^2 - x3*x0,
x1^3 - x4*x3*x0, x4*x1^2 - x5*x0^2) of Multivariate Polynomial Ring
in x5, x4, x3, x2, x1, x0 over Rational Field
sage: S.ideal(True)
[x2 - x0, x3^2 - x5*x0, x5*x3 - x0^2, x4^2 - x3*x1, x5^2 - x3*x0,
x1^3 - x4*x3*x0, x4*x1^2 - x5*x0^2]
sage: S.ideal().gens()  # another way to get the generators
[x2 - x0, x3^2 - x5*x0, x5*x3 - x0^2, x4^2 - x3*x1, x5^2 - x3*x0,
x1^3 - x4*x3*x0, x4*x1^2 - x5*x0^2]
```

— **identity()**

The identity configuration.

INPUT:

None

OUTPUT:

dict (the identity configuration)

EXAMPLES:

```
sage: S = sandlib('generic')
sage: e = S.identity()
sage: x = e & S.max_stable()  # stable addition
sage: x
{1: 2, 2: 2, 3: 1, 4: 1, 5: 1}
sage: x == S.max_stable()
True
```

— **in_degree(v=None)**

The in-degree of a vertex or a list of all in-degrees.

INPUT:

`v` - vertex name or None

OUTPUT:

integer or dict

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.in_degree(2)
2
sage: S.in_degree()
{0: 2, 1: 1, 2: 2, 3: 4, 4: 1, 5: 2}
```

— **invariant_factors()**

The invariant factors of the sandpile group (a finite abelian group).

INPUT:

None

OUTPUT:

list of integers

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.invariant_factors()
[1, 1, 1, 1, 15]
```

— **is_undirected()**

`True` if `(u,v)` is and edge if and only if `(v,u)` is an edges, each edge with the same weight.

INPUT:

None

OUTPUT:

boolean

EXAMPLES:

```
sage: complete_sandpile(4).is_undirected()
True
sage: sandlib('gor').is_undirected()
False
```

— **laplacian()**

The Laplacian matrix of the graph.

INPUT:

None

OUTPUT:

matrix

EXAMPLES:

```
sage: G = sandlib('generic')
sage: G.laplacian()
[ 0  0  0  0  0  0]
[-1  3  0 -1 -1  0]
[-1  0  3 -1  0 -1]
[ 0  0 -1  2  0 -1]
[ 0 -1  0 -1  2  0]
[ 0  0 -1 -1  0  2]
```

NOTES:

The function `laplacian_matrix` should be avoided. It returns the indegree version of the laplacian.

— **max_stable()**

The maximal stable configuration.

INPUT:

None

OUTPUT:

SandpileConfig (the maximal stable configuration)

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.max_stable()
{1: 2, 2: 2, 3: 1, 4: 1, 5: 1}
```

— **max_stable_div()**

The maximal stable divisor.

INPUT:

SandpileDivisor

OUTPUT:

SandpileDivisor (the maximal stable divisor)

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.max_stable_div()
{0: -1, 1: 2, 2: 2, 3: 1, 4: 1, 5: 1}
sage: S.out_degree()
{0: 0, 1: 3, 2: 3, 3: 2, 4: 2, 5: 2}
```

— **max_superstables(verbose=True)**

The maximal superstable configurations. If the underlying graph is undirected, these are the superstables of highest degree. If `verbose` is `False`, the configurations are converted to lists of integers.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

list (of maximal superstables)

EXAMPLES:

```
sage: S=sandlib('riemann-roch2')
sage: S.max_superstables()
[{1: 1, 2: 1, 3: 1}, {1: 0, 2: 0, 3: 2}]
sage: S.superstables(False)
[[0, 0, 0],
 [1, 0, 1],
 [1, 0, 0],
 [0, 1, 1],
 [0, 1, 0],
 [1, 1, 0],
 [0, 0, 1],
 [1, 1, 1],
 [0, 0, 2]]
sage: S.h_vector()
[1, 3, 4, 1]
```

— **min_recurrents(verbose=True)**

The minimal recurrent elements. If the underlying graph is undirected, these are the recurrent elements of least degree. If `verbose is ``False`, the configurations are converted to lists of integers.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

list of SandpileConfig

EXAMPLES:

```
sage: S=sandlib('riemann-roch2')
sage: S.min_recurrents()
[{1: 0, 2: 0, 3: 1}, {1: 1, 2: 1, 3: 0}]
sage: S.min_recurrents(False)
[[0, 0, 1], [1, 1, 0]]
sage: S.recurrents(False)
[[1, 1, 2],
 [0, 1, 1],
 [0, 1, 2],
 [1, 0, 1],
 [1, 0, 2],
 [0, 0, 2],
 [1, 1, 1],
 [0, 0, 1],
 [1, 1, 0]]
sage: [i.deg() for i in S.recurrents()]
[4, 2, 3, 2, 3, 2, 3, 1, 2]
```

— **nonsink_vertices()**

The names of the nonsink vertices.

INPUT:

None

OUTPUT:

None

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.nonsink_vertices()
[1, 2, 3, 4, 5]
```

— **nonspecial_divisors(verbose=True)**

The nonspecial divisors: those divisors of degree `g-1` with empty linear system. The term is only defined for undirected graphs. Here, `g = |E| - |V| + 1` is the genus of the graph. If `verbose` is `False`, the divisors are converted to lists of integers.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

list (of divisors)

EXAMPLES:

```
sage: S = complete_sandpile(4)
sage: ns = S.nonspecial_divisors() # optional - 4ti2
sage: D = ns[0] # optional - 4ti2
sage: D.values() # optional - 4ti2
[-1, 1, 0, 2]
sage: D.deg() # optional - 4ti2
2
sage: [i.effective_div() for i in ns] # optional - 4ti2
[[], [], [], [], [], []]
```

— **num_edges()**

The number of edges.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.size()
15
```

— **num_verts()**

The number of vertices. Note that len(G) returns the number of vertices in G also.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.order()
10

sage: G = graphs.TetrahedralGraph()
sage: len(G)
4
```

— **out_degree(v=None)**

The out-degree of a vertex or a list of all out-degrees.

INPUT:

v (optional) - vertex name

OUTPUT:

integer or dict

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.out_degree(2)
3
sage: S.out_degree()
{0: 0, 1: 3, 2: 3, 3: 2, 4: 2, 5: 2}
```

— **points()**

Generators for the multiplicative group of zeros of the sandpile ideal.

INPUT:

None

OUTPUT:

list of complex numbers

EXAMPLES:

The sandpile group in this example is cyclic, and hence there is a single generator for the group of solutions.

```
sage: S = sandlib('generic')
sage: S.points()
[[e^(4/5*I*pi), 1, e^(2/3*I*pi), e^(-34/15*I*pi), e^(-2/3*I*pi)]]
```

— **postulation()**

The postulation number of the sandpile ideal. This is the largest weight of a superstable configuration of the graph.

INPUT:

None

OUTPUT:

nonnegative integer

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.postulation()
3
```

— **recurrents(verbose=True)**

The list of recurrent configurations. If `verbose` is `False`, the configurations are converted to lists of integers.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

list (of recurrent configurations)

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.recurrents()
[{1: 2, 2: 2, 3: 1, 4: 1, 5: 1}, {1: 2, 2: 2, 3: 0, 4: 1, 5: 1},
{1: 0, 2: 2, 3: 1, 4: 1, 5: 0}, {1: 0, 2: 2, 3: 1, 4: 1, 5: 1},
{1: 1, 2: 2, 3: 1, 4: 1, 5: 1}, {1: 1, 2: 2, 3: 0, 4: 1, 5: 1},
{1: 2, 2: 2, 3: 1, 4: 0, 5: 1}, {1: 2, 2: 2, 3: 0, 4: 0, 5: 1},
{1: 2, 2: 2, 3: 1, 4: 0, 5: 0}, {1: 1, 2: 2, 3: 1, 4: 1, 5: 0},
{1: 1, 2: 2, 3: 1, 4: 0, 5: 0}, {1: 1, 2: 2, 3: 1, 4: 0, 5: 1},
{1: 0, 2: 2, 3: 0, 4: 1, 5: 1}, {1: 2, 2: 2, 3: 1, 4: 1, 5: 0},
{1: 1, 2: 2, 3: 0, 4: 0, 5: 1}]
sage: S.recurrents(verbose=False)
[[2, 2, 1, 1, 1], [2, 2, 0, 1, 1], [0, 2, 1, 1, 0], [0, 2, 1, 1, 1],
[1, 2, 1, 1, 1], [1, 2, 0, 1, 1], [2, 2, 1, 0, 1], [2, 2, 0, 0, 1],
[2, 2, 1, 0, 0], [1, 2, 1, 1, 0], [1, 2, 1, 0, 0], [1, 2, 1, 0, 1],
[0, 2, 0, 1, 1], [2, 2, 1, 1, 0], [1, 2, 0, 0, 1]]
```

— **reduced_laplacian()**

The reduced Laplacian matrix of the graph.

INPUT:

None

OUTPUT:

matrix

EXAMPLES:

```
sage: G = sandlib('generic')
sage: G.laplacian()
[ 0  0  0  0  0  0]
[-1  3  0 -1 -1  0]
[-1  0  3 -1  0 -1]
[ 0  0 -1  2  0 -1]
[ 0 -1  0 -1  2  0]
[ 0  0 -1 -1  0  2]
sage: G.reduced_laplacian()
[ 3  0 -1 -1  0]
[ 0  3 -1  0 -1]
[ 0 -1  2  0 -1]
[-1  0 -1  2  0]
[ 0 -1 -1  0  2]
```

NOTES:

This is the Laplacian matrix with the row and column indexed by the sink vertex removed.

— **reorder_vertices()**

Create a copy of the sandpile but with the vertices ordered according to their distance from the sink, from greatest to least.

INPUT:

None

OUTPUT:

Sandpile

EXAMPLES:

```
sage: S = sandlib('kite')
sage: S.dict()
{0: {}, 1: {0: 1, 2: 1,
3: 1}, 2: {1: 1, 3: 1, 4: 1}, 3: {1: 1, 2: 1, 4: 1}, 4: {2: 1,
3: 1}}
sage: T = S.reorder_vertices()
sage: T.dict()
{0: {1: 1, 2: 1}, 1: {0: 1, 2: 1, 3: 1}, 2: {0: 1, 1: 1, 3: 1},
3: {1: 1, 2: 1, 4: 1}, 4: {}}
```

— **resolution(verbose=False)**

This function computes a minimal free resolution of the homogeneous sandpile ideal. If `verbose` is `True`, then all of the mappings are returned. Otherwise, the resolution is summarized.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

free resolution of the sandpile ideal

EXAMPLES:

```
sage: S = sandlib('gor')
sage: S.resolution()
'R^1 <-- R^5 <-- R^5 <-- R^1'
sage: S.resolution(True)
[
[ x1^2 - x3*x0 x3*x1 - x2*x0  x3^2 - x2*x1  x2*x3 - x0^2  x2^2 - x1*x0],
[ x3  x2   0  x0   0]  [ x2^2 - x1*x0]
[-x1 -x3  x2   0 -x0]  [-x2*x3 + x0^2]
[ x0  x1   0  x2   0]  [-x3^2 + x2*x1]
[  0   0 -x1 -x3  x2]  [x3*x1 - x2*x0]
[  0   0  x0  x1 -x3], [ x1^2 - x3*x0]
]
sage: r = S.resolution(True)
sage: r[0]*r[1]
[0 0 0 0 0]
sage: r[1]*r[2]
[0]
[0]
[0]
[0]
[0]
```

— **ring()**

The ring containing the homogeneous sandpile ideal.

INPUT:

None

OUTPUT:

ring

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.ring()
Multivariate Polynomial Ring in x5, x4, x3, x2, x1, x0 over Rational Field
sage: S.ring().gens()
(x5, x4, x3, x2, x1, x0)
```

NOTES:

The indeterminate xi corresponds to the i-th vertex as listed my the method `vertices`. The term-ordering is degrevlex with indeterminates ordered according to their distance from the sink (larger indeterminates are further from the sink).

— **show(kwds)**

Draws the graph.

INPUT:

`kwds` - arguments passed to the show method for Graph or DiGraph

OUTPUT:

None

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.show()
sage: S.show(graph_border=True, edge_labels=True)
```

— **show3d(kwds)**

Draws the graph.

INPUT:

`kwds` - arguments passed to the show method for Graph or DiGraph

OUTPUT:

None

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.show3d()
```

— **sink()**

The identifier for the sink vertex.

INPUT:

None

OUTPUT:

Object (name for the sink vertex)

EXAMPLES:

```
sage: G = sandlib('generic')
sage: G.sink()
0
sage: H = grid_sandpile(2,2)
sage: H.sink()
'sink'
sage: type(H.sink())
<type 'str'>
```

— **solve()**

Approximations of the complex affine zeros of the sandpile ideal.

INPUT:

None

OUTPUT:

list of complex numbers

EXAMPLES:

```
sage: S = Sandpile({0: {}, 1: {2: 2}, 2: {0: 4, 1: 1}}, 0)
sage: S.solve()
[[-0.707107 + 0.707107*I, 0.707107 - 0.707107*I],
[-0.707107 - 0.707107*I, 0.707107 + 0.707107*I],
[-I, -I], [I, I], [0.707107 + 0.707107*I, -0.707107 - 0.707107*I],
[0.707107 - 0.707107*I, -0.707107 + 0.707107*I], [1, 1], [-1, -1]]
sage: len(_)
8
```

```
sage: S.group_order()
8
```

NOTES:

The solutions form a multiplicative group isomorphic to the sandpile group. Generators for this group are given exactly by `points()`.

— **superstables(verbose=True)**

The list of superstable configurations as dictionaries if `verbose` is `True`, otherwise as lists of integers. The superstables are also known as G-parking functions.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

list (of superstable elements)

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.superstables()
[{1: 0, 2: 0, 3: 0, 4: 0, 5: 0},
 {1: 0, 2: 0, 3: 1, 4: 0, 5: 0},
 {1: 2, 2: 0, 3: 0, 4: 0, 5: 1},
 {1: 2, 2: 0, 3: 0, 4: 0, 5: 0},
 {1: 1, 2: 0, 3: 0, 4: 0, 5: 0},
 {1: 1, 2: 0, 3: 1, 4: 0, 5: 0},
 {1: 0, 2: 0, 3: 0, 4: 1, 5: 0},
 {1: 0, 2: 0, 3: 1, 4: 1, 5: 0},
 {1: 0, 2: 0, 3: 0, 4: 1, 5: 1},
 {1: 1, 2: 0, 3: 0, 4: 0, 5: 1},
 {1: 1, 2: 0, 3: 0, 4: 1, 5: 1},
 {1: 1, 2: 0, 3: 0, 4: 1, 5: 0},
 {1: 2, 2: 0, 3: 1, 4: 0, 5: 0},
 {1: 0, 2: 0, 3: 0, 4: 0, 5: 1},
 {1: 1, 2: 0, 3: 1, 4: 1, 5: 0}]
sage: S.superstables(False)
[[0, 0, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [2, 0, 0, 0, 1],
 [2, 0, 0, 0, 0],
 [1, 0, 0, 0, 0],
 [1, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 1, 1, 0],
 [0, 0, 0, 1, 1],
 [1, 0, 0, 0, 1],
 [1, 0, 0, 1, 1],
 [1, 0, 0, 1, 0],
 [2, 0, 1, 0, 0],
 [0, 0, 0, 0, 1],
 [1, 0, 1, 1, 0]]
```

— **symmetric_recurrents(orbits)**

The list of symmetric recurrent configurations.

INPUT:

`orbits` - list of lists partitioning the vertices

OUTPUT:

list of recurrent configurations

EXAMPLES:

```
sage: S = sandlib('kite')
sage: S.dict()
{0: {},
 1: {0: 1, 2: 1, 3: 1},
 2: {1: 1, 3: 1, 4: 1},
 3: {1: 1, 2: 1, 4: 1},
 4: {2: 1, 3: 1}}
sage: S.symmetric_recurrents([[1],[2,3],[4]])
[{1: 2, 2: 2, 3: 2, 4: 1}, {1: 2, 2: 2, 3: 2, 4: 0}]
sage: S.recurrents()
[{1: 2, 2: 2, 3: 2, 4: 1},
 {1: 2, 2: 2, 3: 2, 4: 0},
 {1: 2, 2: 1, 3: 2, 4: 0},
 {1: 2, 2: 2, 3: 0, 4: 1},
 {1: 2, 2: 0, 3: 2, 4: 1},
 {1: 2, 2: 2, 3: 1, 4: 0},
 {1: 2, 2: 1, 3: 2, 4: 1},
 {1: 2, 2: 2, 3: 1, 4: 1}]
```

NOTES:

The user is responsible for ensuring that the list of orbits comes from a group of symmetries of the underlying graph.

— **unsaturated_ideal()**

The unsaturated, homogeneous sandpile ideal.

INPUT:

None

OUTPUT:

ideal

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.unsaturated_ideal().gens()
[x1^3 - x4*x3*x0, x2^3 - x5*x3*x0, x3^2 - x5*x2, x4^2 - x3*x1, x5^2 - x3*x2]
sage: S.ideal().gens()
[x2 - x0, x3^2 - x5*x0, x5*x3 - x0^2, x4^2 - x3*x1, x5^2 - x3*x0,
x1^3 - x4*x3*x0, x4*x1^2 - x5*x0^2]
```

— **version()**

The version number of Sage Sandpiles.

INPUT:

None

OUTPUT:

string

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.version()
Sage Sandpiles Version 2.3
```

— **vertices(key=None, boundary_first=False)**

A list of the vertices.

INPUT:

- `key` - default: `None` - a function that takes a vertex as its one argument and returns a value that can be used for comparisons in the sorting algorithm.
- `boundary_first` - default: `False` - if `True`, return the boundary vertices first.

OUTPUT:

The vertices of the list.

Warning: There is always an attempt to sort the list before returning the result. However, since any object may be a vertex, there is no guarantee that any two vertices will be comparable. With default objects for vertices (all integers), or when all the vertices are of the same simple type, then there should not be a problem with how the vertices will be sorted. However, if you need to guarantee a total order for the sort, use the `key` argument, as illustrated in the examples below.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

— **zero_config()**

The all-zero configuration.

INPUT:

None

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.zero_config()
{1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
```

— **zero_div()**

The all-zero divisor.

INPUT:

None

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.zero_div()
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0}
```

—

SandpileConfig Summary of methods.

- *+* — Addition of configurations.
- *&* — The stabilization of the sum.
- *greater-equal* — `True` if every component of `self` is at least that of `other`.
- *greater* — `True` if every component of `self` is at least that of `other` and the two configurations are not equal.
- *~* — The stabilized configuration.
- *less-equal* — `True` if every component of `self` is at most that of `other`.
- *less* — `True` if every component of `self` is at most that of `other` and the two configurations are not equal.
- *** — The recurrent element equivalent to the sum.
- *^* — Exponentiation for the *-operator.
- *-* — The additive inverse of the configuration.
- *-* — Subtraction of configurations.
- *add_random()* — Add one grain of sand to a random nonsink vertex.
- *deg()* — The degree of the configuration.
- *dualize()* — The difference between the maximal stable configuration and the configuration.
- *equivalent_recurrent(with_firing_vector=False)* — The equivalent recurrent configuration.
- *equivalent_superstable(with_firing_vector=False)* — The equivalent superstable configuration.
- *fire_script(sigma)* — Fire the script `sigma`, i.e., fire each vertex the indicated number of times.
- *fire_unstable()* — Fire all unstable vertices.
- *fire_vertex(v)* — Fire the vertex `v`.
- *is_recurrent()* — `True` if the configuration is recurrent.
- *is_stable()* — `True` if stable.
- *is_superstable()* — `True` if `config` is superstable.
- *is_symmetric(orbits)* — Is the configuration constant over the vertices in each sublist of `orbits`?
- *order()* — The order of the recurrent element equivalent to `config`.
- *sandpile()* — The configuration's underlying sandpile.
- *show(sink=True,colors=True,heights=False,directed=None,kwds)* — Show the configuration.
- *stabilize(with_firing_vector=False)* — The stabilized configuration. Optionally returns the corresponding firing vector.
- *support()* — Keys of the nonzero values of the dictionary.
- *unstable()* — List of the unstable vertices.
- *values()* — The values of the configuration as a list.

Complete descriptions of SandpileConfig methods.

— **+**

Addition of configurations.

INPUT:

`other` - SandpileConfig

OUTPUT:

sum of `self` and `other`

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [3,2])
sage: c + d
{1: 4, 2: 4}
```

— **&**

The stabilization of the sum.

INPUT:

`other` - SandpileConfig

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(4), 0)
sage: c = SandpileConfig(S, [1,0,0])
sage: c + c  # ordinary addition
{1: 2, 2: 0, 3: 0}
sage: c & c  # add and stabilize
{1: 0, 2: 1, 3: 0}
sage: c*c  # add and find equivalent recurrent
{1: 1, 2: 1, 3: 1}
sage: ~(c + c) == c & c
True
```

— **>=**

`True` if every component of `self` is at least that of `other`.

INPUT:

`other` - SandpileConfig

OUTPUT:

boolean

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [2,3])
sage: e = SandpileConfig(S, [2,0])
sage: c >= c
True
sage: d >= c
True
sage: c >= d
```

```
False
sage: e >= c
False
sage: c >= e
False
```

— >

`True` if every component of `self` is at least that of `other` and the two configurations are not equal.

INPUT:

`other` - SandpileConfig

OUTPUT:

boolean

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [1,3])
sage: c > c
False
sage: d > c
True
sage: c > d
False
```

— ~

The stabilized configuration.

INPUT:

None

OUTPUT:

`SandpileConfig`

EXAMPLES:

```
sage: S = sandlib('generic')
sage: c = S.max_stable() + S.identity()
sage: ~c
{1: 2, 2: 2, 3: 1, 4: 1, 5: 1}
sage: ~c == c.stabilize()
True
```

— <=

`True` if every component of `self` is at most that of `other`.

INPUT:

`other` - SandpileConfig

OUTPUT:

boolean

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [2,3])
sage: e = SandpileConfig(S, [2,0])
sage: c <= c
True
sage: c <= d
True
sage: d <= c
False
sage: c <= e
False
sage: e <= c
False
```

— <

`True` if every component of `self` is at most that of `other` and the two configurations are not equal.

INPUT:

`other` - SandpileConfig

OUTPUT:

boolean

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [2,3])
sage: c < c
False
sage: c < d
True
sage: d < c
False
```

— *

The recurrent element equivalent to the sum.

INPUT:

`other` - SandpileConfig

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(4), 0)
sage: c = SandpileConfig(S, [1,0,0])
sage: c + c  # ordinary addition
{1: 2, 2: 0, 3: 0}
sage: c & c  # add and stabilize
{1: 0, 2: 1, 3: 0}
sage: c*c  # add and find equivalent recurrent
{1: 1, 2: 1, 3: 1}
sage: (c*c).is_recurrent()
True
```

```
sage: c*(-c) == S.identity()
True
```

__ ^

The recurrent element equivalent to the sum of the configuration with itself `k` times. If `k` is negative, do the same for the negation of the configuration. If `k` is zero, return the identity of the sandpile group.

INPUT:

`k` - SandpileConfig

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(4), 0)
sage: c = SandpileConfig(S, [1,0,0])
sage: c^3
{1: 1, 2: 1, 3: 0}
sage: (c + c + c) == c^3
False
sage: (c + c + c).equivalent_recurrent() == c^3
True
sage: c^(-1)
{1: 1, 2: 1, 3: 0}
sage: c^0 == S.identity()
True
```

__ _

The additive inverse of the configuration.

INPUT:

None

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: -c
{1: -1, 2: -2}
```

__ -

Subtraction of configurations.

INPUT:

`other` - SandpileConfig

OUTPUT:

sum of `self` and `other`

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: d = SandpileConfig(S, [3,2])
sage: c - d
{1: -2, 2: 0}
```

— **add_random()**

Add one grain of sand to a random nonsink vertex.

INPUT:

None

OUTPUT:

SandpileConfig

EXAMPLES:

We compute the 'sizes' of the avalanches caused by adding random grains of sand to the maximal stable configuration on a grid graph. The function `stabilize()` returns the firing vector of the stabilization, a dictionary whose values say how many times each vertex fires in the stabilization.

```
sage: S = grid_sandpile(10,10)
sage: m = S.max_stable()
sage: a = []
sage: for i in range(1000):
...       m = m.add_random()
...       m, f = m.stabilize(True)
...       a.append(sum(f.values()))
...
sage: p = list_plot([[log(i+1),log(a.count(i))] for i in [0..max(a)] if a.count(i)])
sage: p.axes_labels(['log(N)','log(D(N))'])
sage: t = text("Distribution of avalanche sizes", (2,2), rgbcolor=(1,0,0))
sage: show(p+t,axes_labels=['log(N)','log(D(N))'])
```

— **deg()**

The degree of the configuration.

INPUT:

None

OUTPUT:

integer

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: c.deg()
3
```

— **dualize()**

The difference between the maximal stable configuration and the configuration.

INPUT:

None

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: S.max_stable()
{1: 1, 2: 1}
sage: c.dualize()
{1: 0, 2: -1}
sage: S.max_stable() - c == c.dualize()
True
```

— **equivalent_recurrent(with_firing_vector=False)**

The recurrent configuration equivalent to the given configuration. Optionally returns the corresponding firing vector.

INPUT:

`with_firing_vector` (optional) - boolean

OUTPUT:

`SandpileConfig` or `[SandpileConfig, firing_vector]`

EXAMPLES:

```
sage: S = sandlib('generic')
sage: c = SandpileConfig(S, [0,0,0,0,0])
sage: c.equivalent_recurrent() == S.identity()
True
sage: x = c.equivalent_recurrent(True)
sage: r = vector([x[0][v] for v in S.nonsink_vertices()])
sage: f = vector([x[1][v] for v in S.nonsink_vertices()])
sage: cv = vector(c.values())
sage: r == cv - f*S.reduced_laplacian()
True
```

NOTES:

Let L be the reduced laplacian, c the initial configuration, r the returned configuration, and f the firing vector. Then r = c - f * L.

— **equivalent_superstable(with_firing_vector=False)**

The equivalent superstable configuration. Optionally returns the corresponding firing vector.

INPUT:

`with_firing_vector` (optional) - boolean

OUTPUT:

`SandpileConfig` or `[SandpileConfig, firing_vector]`

EXAMPLES:

```
sage: S = sandlib('generic')
sage: m = S.max_stable()
sage: m.equivalent_superstable().is_superstable()
True
sage: x = m.equivalent_superstable(True)
sage: s = vector(x[0].values())
sage: f = vector(x[1].values())
```

```
sage: mv = vector(m.values())
sage: s == mv - f*S.reduced_laplacian()
True
```

NOTES:

Let L be the reduced laplacian, c the initial configuration, s the returned configuration, and f the firing vector. Then s = c - f * L.

— **fire_script(sigma)**

Fire the script `sigma`, i.e., fire each vertex the indicated number of times.

INPUT:

`sigma` - SandpileConfig or (list or dict representing a SandpileConfig)

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(4), 0)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.unstable()
[2, 3]
sage: c.fire_script(SandpileConfig(S,[0,1,1]))
{1: 2, 2: 1, 3: 2}
sage: c.fire_script(SandpileConfig(S,[2,0,0])) == c.fire_vertex(1).fire_vertex(1)
True
```

— **fire_unstable()**

Fire all unstable vertices.

INPUT:

None

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(4), 0)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.fire_unstable()
{1: 2, 2: 1, 3: 2}
```

— **fire_vertex(v)**

Fire the vertex `v`.

INPUT:

`v` - vertex

OUTPUT:

SandpileConfig

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: c = SandpileConfig(S, [1,2])
sage: c.fire_vertex(2)
{1: 2, 2: 0}
```

— **is_recurrent()**

`True` if the configuration is recurrent.

INPUT:

None

OUTPUT:

boolean

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.identity().is_recurrent()
True
sage: S.zero_config().is_recurrent()
False
```

— **is_stable()**

`True` if stable.

INPUT:

None

OUTPUT:

boolean

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.max_stable().is_stable()
True
sage: (S.max_stable() + S.max_stable()).is_stable()
False
sage: (S.max_stable() & S.max_stable()).is_stable()
True
```

— **is_superstable()**

`True` if `config` is superstable, i.e., whether its dual is recurrent.

INPUT:

None

OUTPUT:

boolean

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.zero_config().is_superstable()
True
```

— **is_symmetric(orbits)**

This function checks if the values of the configuration are constant over the vertices in each sublist of `orbits`.

INPUT:

`orbits` - list of lists of vertices

OUTPUT:

boolean

EXAMPLES:

```
sage: S = sandlib('kite')
sage: S.dict()
{0: {},
 1: {0: 1, 2: 1, 3: 1},
 2: {1: 1, 3: 1, 4: 1},
 3: {1: 1, 2: 1, 4: 1},
 4: {2: 1, 3: 1}}
sage: c = SandpileConfig(S, [1, 2, 2, 3])
sage: c.is_symmetric([[2,3]])
True
```

— **order()**

The order of the recurrent element equivalent to `config`.

INPUT:

`config` - configuration

OUTPUT:

integer

EXAMPLES:

```
sage: S = sandlib('generic')
sage: [r.order() for r in S.recurrents()]
[3, 3, 5, 15, 15, 15, 5, 15, 15, 5, 15, 5, 15, 1, 15]
```

— **sandpile()**

The configuration's underlying sandpile.

INPUT:

None

OUTPUT:

Sandpile

EXAMPLES:

```
sage: S = sandlib('genus2')
sage: c = S.identity()
sage: c.sandpile()
Digraph on 4 vertices
sage: c.sandpile() == S
True
```

— **show(sink=True,colors=True,heights=False,directed=None,kwds)**

Show the configuration.

INPUT:

- `sink` - whether to show the sink
- `colors` - whether to color-code the amount of sand on each vertex
- `heights` - whether to label each vertex with the amount of sand
- `kwds` - arguments passed to the show method for Graph
- `directed` - whether to draw directed edges

OUTPUT:

None

EXAMPLES:

```
sage: S=sandlib('genus2')
sage: c=S.identity()
sage: S=sandlib('genus2')
sage: c=S.identity()
sage: c.show()
sage: c.show(directed=False)
sage: c.show(sink=False,colors=False,heights=True)
```

— **stabilize(with_firing_vector=False)**

The stabilized configuration. Optionally returns the corresponding firing vector.

INPUT:

`with_firing_vector` (optional) - boolean

OUTPUT:

`SandpileConfig` or `[SandpileConfig, firing_vector]`

EXAMPLES:

```
sage: S = sandlib('generic')
sage: c = S.max_stable() + S.identity()
sage: c.stabilize(True)
[{1: 2, 2: 2, 3: 1, 4: 1, 5: 1}, {1: 1, 2: 5, 3: 7, 4: 1, 5: 6}]
sage: S.max_stable() & S.identity()
{1: 2, 2: 2, 3: 1, 4: 1, 5: 1}
sage: S.max_stable() & S.identity() == c.stabilize()
True
sage: ~c
{1: 2, 2: 2, 3: 1, 4: 1, 5: 1}
```

— **support()**

The input is a dictionary of integers. The output is a list of keys of nonzero values of the dictionary.

INPUT:

None

OUTPUT:

list - support of the config

EXAMPLES:

```
sage: S = sandlib('generic')
sage: c = S.identity()
sage: c.values()
[2, 2, 1, 1, 0]
sage: c.support()
[1, 2, 3, 4]
sage: S.vertices()
[0, 1, 2, 3, 4, 5]
```

— **unstable()**

List of the unstable vertices.

INPUT:

None

OUTPUT:

list of vertices

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(4), 0)
sage: c = SandpileConfig(S, [1,2,3])
sage: c.unstable()
[2, 3]
```

— **values()**

The values of the configuration as a list, sorted in the order of the vertices.

INPUT:

None

OUTPUT:

list of integers

boolean

EXAMPLES:

```
sage: S = Sandpile({'a':[1,'b'], 'b':[1,'a'], 1:['a']},'a')
sage: c = SandpileConfig(S, {'b':1, 1:2})
sage: c
{1: 2, 'b': 1}
sage: c.values()
[2, 1]
sage: S.nonsink_vertices()
[1, 'b']
```

—

SandpileDivisor Summary of methods.

- + — Addition of divisors.
- *greater-equal* — `True` if every component of `self` is at least that of `other`.
- *greater* — `True` if every component of `self` is at least that of `other` and the two divisors are not equal.
- *less-equal* — `True` if every component of `self` is at most that of `other`.

- *less* — `True` if every component of `self` is at most that of `other` and the two divisors are not equal.
- *-* — The additive inverse of the divisor.
- *-* — Subtraction of divisors.
- *add_random()* — Add one grain of sand to a random vertex.
- *betti()* — The Betti numbers for the simplicial complex associated with the divisor.
- *Dcomplex()* — The simplicial complex determined by the supports of the linearly equivalent effective divisors.
- *deg()* — The degree of the divisor.
- *dualize()* — The difference between the maximal stable divisor and the divisor.
- *effective_div(verbose=True)* — All linearly equivalent effective divisors.
- *fire_script(sigma)* — Fire the script `sigma`, i.e., fire each vertex the indicated number of times.
- *fire_unstable()* — Fire all unstable vertices.
- *fire_vertex(v)* — Fire the vertex `v`.
- *is_alive(cycle=False)* — Will the divisor stabilize under repeated firings of all unstable vertices?
- *is_symmetric(orbits)* — Is the divisor constant over the vertices in each sublist of `orbits`?
- *linear_system()* — The complete linear system of a divisor.
- *r_of_D(verbose=False)* — Returns `r(D)`.
- *sandpile()* — The divisor's underlying sandpile.
- *show(heights=True,directed=None,kwds)* — Show the divisor.
- *support()* — List of keys of the nonzero values of the divisor.
- *unstable()* — List of the unstable vertices.
- *values()* — The values of the divisor as a list, sorted in the order of the vertices.

Complete descriptions of SandpileDivisor methods.

— **+**

Addition of divisors.

INPUT:

`other` - SandpileDivisor

OUTPUT:

sum of `self` and `other`

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [3,2,1])
sage: D + E
{0: 4, 1: 4, 2: 4}
```

— **>=**

`True` if every component of `self` is at least that of `other`.

INPUT:

`other` - SandpileDivisor

OUTPUT:

boolean

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [2,3,4])
sage: F = SandpileDivisor(S, [2,0,4])
sage: D >= D
True
sage: E >= D
True
sage: D >= E
False
sage: F >= D
False
sage: D >= F
False
```

— **>**

`True` if every component of `self` is at least that of `other` and the two divisors are not equal.

INPUT:

`other` - SandpileDivisor

OUTPUT:

boolean

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [1,3,4])
sage: D > D
False
sage: E > D
True
sage: D > E
False
```

— **<=**

`True` if every component of `self` is at most that of `other`.

INPUT:

`other` - SandpileDivisor

OUTPUT:

boolean

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [2,3,4])
sage: F = SandpileDivisor(S, [2,0,4])
sage: D <= D
True
sage: D <= E
True
sage: E <= D
False
sage: D <= F
False
sage: F <= D
False
```

— <

`True` if every component of `self` is at most that of `other` and the two divisors are not equal.

INPUT:

`other` - SandpileDivisor

OUTPUT:

boolean

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [2,3,4])
sage: D < D
False
sage: D < E
True
sage: E < D
False
```

— -

The additive inverse of the divisor.

INPUT:

None

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: -D
{0: -1, 1: -2, 2: -3}
```

— -

Subtraction of divisors.

INPUT:

`other` - SandpileDivisor

OUTPUT:

Difference of `self` and `other`

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: E = SandpileDivisor(S, [3,2,1])
sage: D - E
{0: -2, 1: 0, 2: 2}
```

— **add_random()**

Add one grain of sand to a random vertex.

INPUT:

None

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = sandlib('generic')
sage: S.zero_div().add_random()  #random
{0: 0, 1: 0, 2: 0, 3: 1, 4: 0, 5: 0}
```

— **betti()**

The Betti numbers for the simplicial complex associated with the divisor.

INPUT:

None

OUTPUT:

dictionary of integers

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [2,0,1])
sage: D.betti() # optional - 4ti2
{0: 1, 1: 1}
```

— **Dcomplex()**

The simplicial complex determined by the supports of the linearly equivalent effective divisors.

INPUT:

None

OUTPUT:

simplicial complex

EXAMPLES:

```
sage: S = sandlib('generic')
sage: p = SandpileDivisor(S, [0,1,2,0,0,1]).Dcomplex() # optional - 4ti2
sage: p.homology() # optional - 4ti2
{0: 0, 1: Z x Z, 2: 0, 3: 0}
```

```
sage: p.f_vector() # optional - 4ti2
[1, 6, 15, 9, 1]
sage: p.betti() # optional - 4ti2
{0: 1, 1: 2, 2: 0, 3: 0}
```

— **deg()**

The degree of the divisor.

INPUT:

None

OUTPUT:

integer

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.deg()
6
```

— **dualize()**

The difference between the maximal stable divisor and the divisor.

INPUT:

None

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.dualize()
{0: 0, 1: -1, 2: -2}
sage: S.max_stable_div() - D == D.dualize()
True
```

— **effective_div(verbose=True)**

All linearly equivalent effective divisors. If `verbose` is `False`, the divisors are converted to lists of integers.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

list (of divisors)

EXAMPLES:

```
sage: S = sandlib('generic')
sage: D = SandpileDivisor(S, [0,0,0,0,0,2]) # optional - 4ti2
sage: D.effective_div() # optional - 4ti2
[{0: 1, 1: 0, 2: 0, 3: 1, 4: 0, 5: 0},
 {0: 0, 1: 0, 2: 1, 3: 1, 4: 0, 5: 0},
 {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 2}]
```

```
sage: D.effective_div(False) # optional - 4ti2
[[1, 0, 0, 1, 0, 0], [0, 0, 1, 1, 0, 0], [0, 0, 0, 0, 0, 2]]
```

— **fire_script(sigma)**

Fire the script `sigma`, i.e., fire each vertex the indicated number of times.

INPUT:

`sigma` - SandpileDivisor or (list or dict representing a SandpileDivisor)

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.unstable()
[1, 2]
sage: D.fire_script([0,1,1])
{0: 3, 1: 1, 2: 2}
sage: D.fire_script(SandpileDivisor(S,[2,0,0])) == D.fire_vertex(0).fire_vertex(0)
True
```

— **fire_unstable()**

Fire all unstable vertices.

INPUT:

None

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.fire_unstable()
{0: 3, 1: 1, 2: 2}
```

— **fire_vertex(v)**

Fire the vertex `v`.

INPUT:

`v` - vertex

OUTPUT:

SandpileDivisor

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.fire_vertex(1)
{0: 2, 1: 0, 2: 4}
```

— **is_alive(cycle=False)**

Will the divisor stabilize under repeated firings of all unstable vertices? Optionally returns the resulting cycle.

INPUT:

`cycle` (optional) - boolean

OUTPUT:

boolean or optionally, a list of SandpileDivisors

EXAMPLES:

```
sage: S = complete_sandpile(4)
sage: D = SandpileDivisor(S, {0: 4, 1: 3, 2: 3, 3: 2})
sage: D.is_alive()
True
sage: D.is_alive(True)
[{0: 4, 1: 3, 2: 3, 3: 2}, {0: 3, 1: 2, 2: 2, 3: 5}, {0: 1, 1: 4, 2: 4, 3: 3}]
```

— **is_symmetric(orbits)**

This function checks if the values of the divisor are constant over the vertices in each sublist of `orbits`.

INPUT:

- `orbits` - list of lists of vertices

OUTPUT:

boolean

EXAMPLES:

```
sage: S = sandlib('kite')
sage: S.dict()
{0: {},
 1: {0: 1, 2: 1, 3: 1},
 2: {1: 1, 3: 1, 4: 1},
 3: {1: 1, 2: 1, 4: 1},
 4: {2: 1, 3: 1}}
sage: D = SandpileDivisor(S, [2,1, 2, 2, 3])
sage: D.is_symmetric([[0,2,3]])
True
```

— **linear_system()**

The complete linear system of a divisor.

INPUT: None

OUTPUT:

dict - `{num_homog: int, homog:list, num_inhomog:int, inhomog:list}`

EXAMPLES:

```
sage: S = sandlib('generic')
sage: D = SandpileDivisor(S, [0,0,0,0,0,2])
sage: D.linear_system() # optional - 4ti2
{'inhomog': [[0, 0, -1, -1, 0, -2], [0, 0, 0, 0, 0, -1], [0, 0, 0, 0, 0, 0]],
'num_inhomog': 3, 'num_homog': 2, 'homog': [[1, 0, 0, 0, 0, 0],
[-1, 0, 0, 0, 0, 0]]}
```

NOTES:

If L is the Laplacian, an arbitrary v such that v * L>= -D has the form v = w + t where w is in `inhomg` and t is in the integer span of `homog` in the output of `linear_system(D)`.

WARNING:

This method requires 4ti2.

— **r_of_D(verbose=False)**

Returns `r(D)` and, if `verbose` is `True`, an effective divisor `F` such that `|D - F|` is empty.

INPUT:

`verbose` (optional) - boolean

OUTPUT:

integer `r(D)` or tuple (integer `r(D)`, divisor `F`)

EXAMPLES:

```
sage: S = sandlib('generic')
sage: D = SandpileDivisor(S, [0,0,0,0,0,4]) # optional - 4ti2
sage: E = D.r_of_D(True) # optional - 4ti2
sage: E # optional - 4ti2
(1, {0: 0, 1: 1, 2: 0, 3: 1, 4: 0, 5: 0})
sage: F = E[1] # optional - 4ti2
sage: (D - F).values() # optional - 4ti2
[0, -1, 0, -1, 0, 4]
sage: (D - F).effective_div() # optional - 4ti2
[]
sage: SandpileDivisor(S, [0,0,0,0,0,-4]).r_of_D(True) # optional - 4ti2
(-1, {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: -4})
```

— **sandpile()**

The divisor's underlying sandpile.

INPUT:

None

OUTPUT:

Sandpile

EXAMPLES:

```
sage: S = sandlib('genus2')
sage: D = SandpileDivisor(S,[1,-2,0,3])
sage: D.sandpile()
Digraph on 4 vertices
sage: D.sandpile() == S
True
```

— **show(heights=True,directed=None,kwds)**

Show the divisor.

INPUT:

- `heights` - whether to label each vertex with the amount of sand
- `kwds` - arguments passed to the show method for Graph

- `directed` - whether to draw directed edges

OUTPUT:

None

EXAMPLES:

```
sage: S = sandlib('genus2')
sage: D = SandpileDivisor(S,[1,-2,0,2])
sage: D.show(graph_border=True,vertex_size=700,directed=False)
```

— **support()**

List of keys of the nonzero values of the divisor.

INPUT:

None

OUTPUT:

list - support of the divisor

EXAMPLES:

```
sage: S = sandlib('generic')
sage: c = S.identity()
sage: c.values()
[2, 2, 1, 1, 0]
sage: c.support()
[1, 2, 3, 4]
sage: S.vertices()
[0, 1, 2, 3, 4, 5]
```

— **unstable()**

List of the unstable vertices.

INPUT:

None

OUTPUT:

list of vertices

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(3), 0)
sage: D = SandpileDivisor(S, [1,2,3])
sage: D.unstable()
[1, 2]
```

— **values()**

The values of the divisor as a list, sorted in the order of the vertices.

INPUT:

None

OUTPUT:

list of integers

boolean

EXAMPLES:

```
sage: S = Sandpile({'a':[1,'b'], 'b':[1,'a'], 1:['a']},'a')
sage: D = SandpileDivisor(S, {'a':0, 'b':1, 1:2})
sage: D
{'a': 0, 1: 2, 'b': 1}
sage: D.values()
[2, 0, 1]
sage: S.vertices()
[1, 'a', 'b']
```

Other

- *admissible_partitions(S, k)* — Partitions of the vertices into `k` parts, each of which is connected.
- *aztec_sandpile(n)* — The aztec diamond graph.
- *complete_sandpile(n)* — Sandpile on the complete graph.
- *firing_graph(S, eff)* — The firing graph.
- *firing_vector(S,D,E)* — The firing vector taking divisor `D` to divisor `E`.
- *glue_graphs(g,h,glue_g,glue_h)* — Glue two sandpiles together.
- *grid_sandpile(m,n)* — The $m \times n$ grid sandpile.
- *min_cycles(G,v)* — The minimal length cycles in the digraph `G` starting at vertex `v`.
- *parallel_firing_graph(S,eff)* — The parallel-firing graph.
- *partition_sandpile(S,p)* — Sandpile formed with vertices consisting of parts of an admissible partition.
- *random_digraph(num_verts,p=1/2,directed=True,weight_max=1)* — A random directed graph.
- *random_DAG(num_verts,p=1/2,weight_max=1)* — A random directed acyclic graph.
- *random_tree(n,d)* — Random tree sandpile.
- *sandlib(selector=None)* — A collection of sandpiles.
- *triangle_sandpile(n)* — The triangle sandpile.
- *wilmes_algorithm(M)* — Find matrix with the same integer row span as `M` that is the reduced Laplacian of a digraph.

Complete descriptions of methods. admissible_partitions(S, k)

The partitions of the vertices of `S` into `k` parts, each of which is connected.

INPUT:

`S` - Sandpile `k` - integer

OUTPUT:

list of partitions

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(4), 0)
sage: P = [admissible_partitions(S, i) for i in [2,3,4]]
sage: P
[[{0}, {1, 2, 3}],
```

```
  {{0, 2, 3}, {1}},
  {{0, 1, 3}, {2}},
  {{0, 1, 2}, {3}},
  {{0, 1}, {2, 3}},
  {{0, 3}, {1, 2}}],
 [{{0}, {1}, {2, 3}},
  {{0}, {1, 2}, {3}},
  {{0, 3}, {1}, {2}},
  {{0, 1}, {2}, {3}}],
 [{{0}, {1}, {2}, {3}}]]
sage: for p in P: # long time
...     sum([partition_sandpile(S, i).betti(verbose=false)[-1] for i in p]) # long time
6
8
3
sage: S.betti()  # long time
            0     1     2     3
------------------------------
    0:      1     -     -     -
    1:      -     6     8     3
------------------------------
total:      1     6     8     3
```

— **aztec(n)**

The aztec diamond graph.

INPUT:

n - integer

OUTPUT:

dictionary for the aztec diamond graph

EXAMPLES:

```
sage: aztec_sandpile(2)
{'sink': {(3/2, 1/2): 2, (-1/2, -3/2): 2, (-3/2, 1/2): 2, (1/2, 3/2): 2,
(1/2, -3/2): 2, (-3/2, -1/2): 2, (-1/2, 3/2): 2, (3/2, -1/2): 2},
(1/2, 3/2): {(-1/2, 3/2): 1, (1/2, 1/2): 1, 'sink': 2}, (1/2, 1/2):
{(1/2, -1/2): 1, (3/2, 1/2): 1, (1/2, 3/2): 1, (-1/2, 1/2): 1},
(-3/2, 1/2): {(-3/2, -1/2): 1, 'sink': 2, (-1/2, 1/2): 1}, (-1/2, -1/2):
{(-3/2, -1/2): 1, (1/2, -1/2): 1, (-1/2, -3/2): 1, (-1/2, 1/2): 1},
(-1/2, 1/2): {(-3/2, 1/2): 1, (-1/2, -1/2): 1, (-1/2, 3/2): 1,
(1/2, 1/2): 1}, (-3/2, -1/2): {(-3/2, 1/2): 1, (-1/2, -1/2): 1, 'sink': 2},
(3/2, 1/2): {(1/2, 1/2): 1, (3/2, -1/2): 1, 'sink': 2}, (-1/2, 3/2):
{(1/2, 3/2): 1, 'sink': 2, (-1/2, 1/2): 1}, (1/2, -3/2): {(1/2, -1/2): 1,
(-1/2, -3/2): 1, 'sink': 2}, (3/2, -1/2): {(3/2, 1/2): 1, (1/2, -1/2): 1,
'sink': 2}, (1/2, -1/2): {(1/2, -3/2): 1, (-1/2, -1/2): 1, (1/2, 1/2): 1,
(3/2, -1/2): 1}, (-1/2, -3/2): {(-1/2, -1/2): 1, 'sink': 2, (1/2, -3/2): 1}}
sage: Sandpile(aztec_sandpile(2),'sink').group_order()
4542720
```

NOTES:

This is the aztec diamond graph with a sink vertex added. Boundary vertices have edges to the sink so that each vertex has degree 4.

— **complete_sandpile(n)**

The sandpile on the complete graph with n vertices.

INPUT:

`n` - positive integer

OUTPUT:

Sandpile

EXAMPLES:

```
sage: K = complete_sandpile(5)
sage: K.betti(verbose=False) # long time
[1, 15, 50, 60, 24]
```

— **firing_graph(S, eff)**

Creates a digraph with divisors as vertices and edges between two divisors `D` and `E` if firing a single vertex in `D` gives `E`.

INPUT:

`S` - sandpile `eff` - list of divisors

OUTPUT:

DiGraph

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(6),0)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: eff = D.effective_div() # optional - 4ti2
sage: firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01) # optional 4ti2
```

— **firing_vector(S,D,E)**

If `D` and `E` are linearly equivalent divisors, find the firing vector taking `D` to `E`.

INPUT:

- `S` -Sandpile
- `D`, `E` - tuples (representing linearly equivalent divisors)

OUTPUT:

tuple (representing a firing vector from `D` to `E`)

EXAMPLES:

```
sage: S = complete_sandpile(4)
sage: D = SandpileDivisor(S, {0: 0, 1: 0, 2: 8, 3: 0})
sage: E = SandpileDivisor(S, {0: 2, 1: 2, 2: 2, 3: 2})
sage: v = firing_vector(S, D, E)
sage: v
(0, 0, 2, 0)
sage: vector(D.values()) - S.laplacian()*vector(v) == vector(E.values())
True
```

The divisors must be linearly equivalent:

```
sage: S = complete_sandpile(4)
sage: D = SandpileDivisor(S, {0: 0, 1: 0, 2: 8, 3: 0})
sage: firing_vector(S, D, S.zero_div())
Error. Are the divisors linearly equivalent?
```

— **glue_graphs(g,h,glue_g,glue_h)**

Glue two graphs together.

INPUT:

- g, h - dictionaries for directed multigraphs
- glue_h, glue_g - dictionaries for a vertex

OUTPUT:

dictionary for a directed multigraph

EXAMPLES:

```
sage: x = {0: {}, 1: {0: 1}, 2: {0: 1, 1: 1}, 3: {0: 1, 1: 1, 2: 1}}
sage: y = {0: {}, 1: {0: 2}, 2: {1: 2}, 3: {0: 1, 2: 1}}
sage: glue_x = {1: 1, 3: 2}
sage: glue_y = {0: 1, 1: 2, 3: 1}
sage: z = glue_graphs(x,y,glue_x,glue_y)
sage: z
{0: {}, 'y2': {'y1': 2}, 'y1': {0: 2}, 'x2': {'x0': 1, 'x1': 1},
'x3': {'x2': 1, 'x0': 1, 'x1': 1}, 'y3': {0: 1, 'y2': 1},
'x1': {'x0': 1}, 'x0': {0: 1, 'x3': 2, 'y3': 1, 'x1': 1, 'y1': 2}}
sage: S = Sandpile(z,0)
sage: S.h_vector()
[1, 6, 17, 31, 41, 41, 31, 17, 6, 1]
sage: S.resolution() # long time
'R^1 <-- R^7 <-- R^21 <-- R^35 <-- R^35 <-- R^21 <-- R^7 <-- R^1'
```

NOTES:

This method makes a dictionary for a graph by combining those for g and h. The sink of g is replaced by a vertex that is connected to the vertices of g as specified by glue_g the vertices of h as specified in glue_h. The sink of the glued graph is 0.

Both glue_g and glue_h are dictionaries with entries of the form v:w where v is the vertex to be connected to and w is the weight of the connecting edge.

— **grid_sandpile(m,n)**

The mxn grid sandpile. Each nonsink vertex has degree 4.

INPUT: m, n - positive integers

OUTPUT: dictionary for a sandpile with sink named sink.

EXAMPLES:

```
sage: grid_sandpile(3,4).dict()
{(1, 2): {(1, 1): 1, (1, 3): 1, 'sink': 1, (2, 2): 1},
(3, 2): {(3, 3): 1, (3, 1): 1, 'sink': 1, (2, 2): 1},
(1, 3): {(1, 2): 1, (2, 3): 1, 'sink': 1, (1, 4): 1},
(3, 3): {(2, 3): 1, (3, 2): 1, (3, 4): 1, 'sink': 1},
(3, 1): {(3, 2): 1, 'sink': 2, (2, 1): 1},
(1, 4): {(1, 3): 1, (2, 4): 1, 'sink': 2},
(2, 4): {(2, 3): 1, (3, 4): 1, 'sink': 1, (1, 4): 1},
(2, 3): {(3, 3): 1, (1, 3): 1, (2, 4): 1, (2, 2): 1},
(2, 1): {(1, 1): 1, (3, 1): 1, 'sink': 1, (2, 2): 1},
(2, 2): {(1, 2): 1, (3, 2): 1, (2, 3): 1, (2, 1): 1},
(3, 4): {(2, 4): 1, (3, 3): 1, 'sink': 2},
(1, 1): {(1, 2): 1, 'sink': 2, (2, 1): 1},
'sink': {}}
```

```
sage: grid_sandpile(3,4).group_order()
4140081
```

— **min_cycles(G,v)**

Minimal length cycles in the digraph G starting at vertex v.

INPUT:

G - DiGraph v - vertex of G

OUTPUT:

list of lists of vertices

EXAMPLES:

```
sage: T = sandlib('gor')
sage: [min_cycles(T, i) for i in T.vertices()]
[[], [[1, 3]], [[2, 3, 1], [2, 3]], [[3, 1], [3, 2]]]
```

— **parallel_firing_graph(S,eff)**

Creates a digraph with divisors as vertices and edges between two divisors D and E if firing all unstable vertices in D gives E.

INPUT:

S - Sandpile eff - list of divisors

OUTPUT:

DiGraph

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(6),0)
sage: D = SandpileDivisor(S, [1,1,1,1,2,0])
sage: eff = D.effective_div() # optional - 4ti2
sage: parallel_firing_graph(S,eff).show3d(edge_size=.005,vertex_size=0.01) # optional - 4ti2
```

— **partition_sandpile(S,p)**

Each set of vertices in p is regarded as a single vertex, with and edge between A and B if some element of A is connected by an edge to some element of B in S.

INPUT:

S - Sandpile p - partition of the vertices of S

OUTPUT:

Sandpile

EXAMPLES:

```
sage: S = Sandpile(graphs.CycleGraph(4), 0)
sage: P = [admissible_partitions(S, i) for i in [2,3,4]]
sage: for p in P:  #long time
...       sum([partition_sandpile(S, i).betti(verbose=false)[-1] for i in p]) # long time
6
8
3
sage: S.betti() # long time
           0     1     2     3
-------------------------------
```

```
    0:     1     -     -     -
    1:     -     6     8     3
------------------------------
total:     1     6     8     3
```

— **random_digraph(num_verts,p=1/2,directed=True,weight_max=1)**

A random weighted digraph with a directed spanning tree rooted at 0. If `directed = False`, the only difference is that if (i, j, w) is an edge with tail i, head j, and weight w, then (j, i, w) appears also. The result is returned as a Sage digraph.

INPUT:

- `num_verts` - number of vertices
- `p` - probability edges occur
- `directed` - True if directed
- `weight_max` - integer maximum for random weights

OUTPUT:

random graph

EXAMPLES:

```
sage: g = random_digraph(6,0.2,True,3)
sage: S = Sandpile(g,0)
sage: S.show(edge_labels=True)
```

— **random_DAG(num_verts,p=1/2,weight_max=1)**

Returns a random directed acyclic graph with `num_verts` vertices. The method starts with the sink vertex and adds vertices one at a time. Each vertex is connected only to only previously defined vertices, and the probability of each possible connection is given by the argument `p`. The weight of an edge is a random integer between `1` and `weight_max`.

INPUT:

- `num_verts` - positive integer
- `p` - number between 0 and 1
- `weight_max` – integer greater than 0

OUTPUT:

directed acyclic graph with sink 0

EXAMPLES:

```
sage: S = random_DAG(5, 0.3)
```

— **random_tree(n,d)**

Returns a random undirected tree with `n` nodes, no node having degree higher than `d`.

INPUT:

`n`, `d` - integers

OUTPUT:

Graph

EXAMPLES:

```
sage: T = random_tree(15,3)
sage: T.show()
sage: S = Sandpile(T,0)
sage: U = S.reorder_vertices()
sage: Graph(U).show()
```

— **sandlib(selector=None)**

The sandpile identified by `selector`. If no argument is given, a description of the sandpiles in the sandlib is printed.

INPUT:

`selector` - identifier or None

OUTPUT:

sandpile or description

EXAMPLES:

```
sage: sandlib()
  Sandpiles in the sandlib:
     kite : generic undirected graphs with 5 vertices
     generic : generic digraph with 6 vertices
     genus2 : Undirected graph of genus 2
     ci1 : complete intersection, non-DAG but  equivalent to a DAG
     riemann-roch1 : directed graph with postulation 9 and 3 maximal weight superstables
     riemann-roch2 : directed graph with a superstable not majorized by a maximal superstable
     gor : Gorenstein but not a complete intersection
```

— **triangle(n)**

A triangular sandpile. Each nonsink vertex has out-degree six. The vertices on the boundary of the triangle are connected to the sink.

INPUT:

`n` - int

OUTPUT:

Sandpile

EXAMPLES:

```
sage: T = triangle_sandpile(5)
sage: T.group_order()
135418115000
```

— **wilmes_algorithm(M)**

Computes an integer matrix `L` with the same integer row span as `M` and such that `L` is the reduced laplacian of a directed multigraph.

INPUT:

`M` - square integer matrix of full rank

OUTPUT:

`L` - integer matrix

EXAMPLES:

```
sage: P = matrix([[2,3,-7,-3],[5,2,-5,5],[8,2,5,4],[-5,-9,6,6]])
sage: wilmes_algorithm(P)
[ 1642   -13 -1627    -1]
[   -1  1980 -1582  -397]
[    0    -1  1650 -1649]
[    0     0 -1658  1658]
```

NOTES:

The algorithm is due to John Wilmes.

Help

Documentation for each method is available through the Sage online help system:

```
sage: SandpileConfig.fire_vertex?
Base Class:     <type 'instancemethod'>
String Form:    <unbound method SandpileConfig.fire_vertex>
Namespace:      Interactive
File:           /usr/local/sage-4.7/local/lib/python2.6/site-packages/sage/sandpiles/sandpile.py
Definition:     SandpileConfig.fire_vertex(self, v)
Docstring:
       Fire the vertex ``v``.

       INPUT:

       ``v`` - vertex

       OUTPUT:

       SandpileConfig

       EXAMPLES:

          sage: S = Sandpile(graphs.CycleGraph(3), 0)
          sage: c = SandpileConfig(S, [1,2])
          sage: c.fire_vertex(2)
          {1: 2, 2: 0}
```

Note: An alternative to `SandpileConfig.fire_vertex?` in the preceding code example would be `c.fire_vertex?`, if `c` is any SandpileConfig.

General Sage documentation can be found at http://sagemath.org/doc/.

Contact

Please contact davidp@reed.edu with questions, bug reports, and suggestions for additional features and other improvements.

6.1.4 Group Theory and Sage

Author: Robert A. Beezer, University of Puget Sound

Changelog:

- 2009/01/30 Version 1.0, first complete release
- 2009/03/03 Version 1.1, added cyclic group size interact
- 2010/03/10 Version 1.3, dropped US on license, some edits.

This compilation collects Sage commands that are useful for a student in an introductory course on group theory. It is not intended to teach Sage or to teach group theory. (There are many introductory texts on group theory and more information on Sage can be found via www.sagemath.org.) Rather, by presenting commands roughly in the order a student would learn the corresponding mathematics they might be encouraged to experiment and learn more about mathematics and learn more about Sage. Not coincidentally, when Sage was the acronym SAGE, the "E" in Sage stood for "Experimentation."

This guide is also distributed in PDF format and as a Sage worksheet. The worksheet version can be imported into the Sage notebook environment running in a web browser, and then the displayed chunks of code may be executed by Sage if one clicks on the small "evaluate" link below each cell, for a fully interactive experience. A PDF and Sage worksheet versions of this tutorial are available at http://abstract.ups.edu/sage-aata.html.

Basic properties of the integers

Integer division

The command `a % b` will return the remainder upon division of a by b. In other words, the value is the unique integer r such that:

1. $0 \leq r < b$; and
2. $a = bq + r$ for some integer q (the quotient).

Then $(a - r)/b$ will equal q. For example:

```
sage: r = 14 % 3
sage: q = (14 - r) / 3
sage: r, q
(2, 4)
```

will return `2` for the value of `r` and `4` for the value of `q`. Note that the "`/`" is *integer* division, where any remainder is cast away and the result is always an integer. So, for example, `14 / 3` will again equal `4`, not `4.66666`.

Greatest common divisor

The greatest common divisor of a and b is obtained with the command `gcd(a,b)`, where in our first uses, a and b are integers. Later, a and b can be other objects with a notion of divisibility and "greatness," such as polynomials. For example:

```
sage: gcd(2776, 2452)
4
```

Extended greatest common divisor

The command `xgcd(a, b)` ("eXtended GCD") returns a triple where the first element is the greatest common divisor of a and b (as with the `gcd(a, b)` command above), but the next two elements are the values of r and s such that $ra + sb = \gcd(a, b)$. For example, `xgcd(633, 331)` returns `(1, 194, -371)`. Portions of the triple can be extracted using `[ ]` to access the entries of the triple, starting with the first as number `0`. For example, the following should return the result `True` (even if you change the values of `a` and `b`). Studying this block of code will

go a long way towards helping you get the most out of Sage's output. (Note that "=" is how a value is assigned to a variable, while as in the last line, "==" is how we determine equality of two items.)

```
sage: a = 633
sage: b = 331
sage: extended = xgcd(a, b)
sage: g = extended[0]
sage: r = extended[1]
sage: s = extended[2]
sage: g == r*a + s*b
True
```

Divisibility

A remainder of zero indicates divisibility. So `(a % b) == 0` will return `True` if b divides a, and will otherwise return `False`. For example, `(9 % 3) == 0` is `True`, but `(9 % 4) == 0` is `False`. Try predicting the output of the following before executing it in Sage.

```
sage: answer1 = ((20 % 5) == 0)
sage: answer2 = ((17 % 4) == 0)
sage: answer1, answer2
(True, False)
```

Factoring

As promised by the Fundamental Theorem of Arithmetic, `factor(a)` will return a unique expression for a as a product of powers of primes. It will print in a nicely-readable form, but can also be manipulated with Python as a list of pairs (p_i, e_i) containing primes as bases, and their associated exponents. For example:

```
sage: factor(2600)
2^3 * 5^2 * 13
```

If you just want the prime divisors of an integer, then use the `prime_divisors(a)` command, which will return a list of all the prime divisors of a. For example:

```
sage: prime_divisors(2600)
[2, 5, 13]
```

We can strip off other pieces of the prime decomposition using two levels of `[ ]`. This is another good example to study in order to learn about how to drill down into Python lists.

```
sage: n = 2600
sage: decomposition = factor(n)
sage: print n, "decomposes as", decomposition
2600 decomposes as 2^3 * 5^2 * 13
sage: secondterm = decomposition[1]
sage: print "Base and exponent (pair) for second prime:", secondterm
Base and exponent (pair) for second prime: (5, 2)
sage: base = secondterm[0]
sage: exponent = secondterm[1]
sage: print "Base is", base
Base is 5
sage: print "Exponent is", exponent
Exponent is 2
sage: thirdbase = decomposition[2][0]
```

```
sage: thirdexponent = decomposition[2][1]
sage: print "Base of third term is", thirdbase, "with exponent", thirdexponent
Base of third term is 13 with exponent  1
```

With a bit more work, the `factor()` command can be used to factor more complicated items, such as polynomials.

Multiplicative inverse, modular arithmetic

The command `inverse_mod(a, n)` yields the multiplicative inverse of a mod n (or an error if it doesn't exist). For example:

```
sage: inverse_mod(352, 917)
508
```

(As a check, find the integer m such that `352*508 == m*917+1`.) Then try

```
sage: inverse_mod(4, 24)
Traceback (most recent call last):
...
ZeroDivisionError: Inverse does not exist.
```

and explain the result.

Powers with modular arithmetic

The command `power_mod(a, m, n)` yields a^m mod n. For example:

```
sage: power_mod(15, 831, 23)
10
```

If $m = -1$, then this command will duplicate the function of `inverse_mod()`.

Euler ϕ-function

The command `euler_phi(n)` will return the number of positive integers less than n and relatively prime to n (i.e. having greatest common divisor with n equal to 1). For example:

```
sage: euler_phi(345)
176
```

Experiment by running the following code several times:

```
sage: m = random_prime(10000)
sage: n = random_prime(10000)
sage: euler_phi(m*n) == euler_phi(m) * euler_phi(n)
True
```

Feel a conjecture coming on? Can you generalize this result?

Primes

The command `is_prime(a)` returns `True` or `False` depending on if a is prime or not. For example,

```
sage: is_prime(117371)
True
```

while

```
sage: is_prime(14547073)
False
```

since $14547073 = 1597 * 9109$ (as you could determine with the `factor()` command).

The command `random_prime(a, True)` will return a random prime between 2 and a. Experiment with:

```
sage: p = random_prime(10^21, True)
sage: is_prime(p)
True
```

(Replacing `True` by `False` will speed up the search, but there will be a very small probability the result will not be prime.)

The command `prime_range(a, b)` returns an ordered list of all the primes from a to $b - 1$, inclusive. For example,

```
sage: prime_range(500, 550)
[503, 509, 521, 523, 541, 547]
```

The commands `next_prime(a)` and `previous_prime(a)` are other ways to get a single prime number of a desired size. Give them a try.

Permutation groups

A good portion of Sage's support for group theory is based on routines from GAP (Groups, Algorithms, and Programming at http://www.gap-system.org. Groups can be described in many different ways, such as sets of matrices or sets of symbols subject to a few defining relations. A very concrete way to represent groups is via permutations (one-to-one and onto functions of the integers 1 through n), using function composition as the operation in the group. Sage has many routines designed to work with groups of this type and they are also a good way for those learning group theory to gain experience with the basic ideas of group theory. For both these reasons, we will concentrate on these types of groups.

Writing permutations

Sage uses "disjoint cycle notation" for permutations, see any introductory text on group theory (such as Judson, Section 4.1) for more on this. Composition occurs *left to right*, which is not what you might expect and is exactly the reverse of what Judson and many others use. (There are good reasons to support either direction, you just need to be certain you know which one is in play.) There are two ways to write the permutation $\sigma = (1\,3)(2\,5\,4)$:

1. As a text string (include quotes): `"(1,3) (2,5,4)"`
2. As a Python list of "tuples": `[(1,3), (2,5,4)]`

Groups

Sage knows many popular groups as sets of permutations. More are listed below, but for starters, the full "symmetric group" of all possible permutations of 1 through n can be built with the command `SymmetricGroup(n)`.

Permutation elements Elements of a group can be created, and composed, as follows

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,3) (2,5,4)")
sage: rho = G([(2,4), (1,5)])
sage: rho^(-1) * sigma * rho
(1,2,4)(3,5)
```

Available functions for elements of a permutation group include finding the order of an element, i.e. for a permutation σ the order is the smallest power of k such that σ^k equals the identity element $()$. For example:

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,3) (2,5,4)")
sage: sigma.order()
6
```

The sign of the permutation σ is defined to be 1 for an even permutation and -1 for an odd permutation. For example:

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,3) (2,5,4)")
sage: sigma.sign()
-1
```

since σ is an odd permutation.

Many more available functions that can be applied to a permutation can be found via "tab-completion." With `sigma` defined as an element of a permutation group, in a Sage cell, type `sigma.` (Note the "`.`") and then press the tab key. You will get a list of available functions (you may need to scroll down to see the whole list). Experiment and explore! It is what Sage is all about. You really cannot break anything.

Creating groups

This is an annotated list of some small well-known permutation groups that can be created simply in Sage. (You can find more in the source code file

`SAGE_ROOT/devel/sage/sage/groups/perm_gps/permgroup_named.py`

- `SymmetricGroup(n)`: All $n!$ permutations on n symbols.
- `DihedralGroup(n)`: Symmetries of an n-gon. Rotations and flips, $2n$ in total.
- `CyclicPermutationGroup(n)`: Rotations of an n-gon (no flips), n in total.
- `AlternatingGroup(n)`: Alternating group on n symbols having $n!/2$ elements.
- `KleinFourGroup()`: The non-cyclic group of order 4.

Group functions

Individual elements of permutation groups are important, but we primarily wish to study groups as objects on their own. So a wide variety of computations are available for groups. Define a group, for example

```
sage: H = DihedralGroup(6)
sage: H
Dihedral group of order 12 as a permutation group
```

and then a variety of functions become available.

After trying the examples below, experiment with tab-completion. Having defined `H`, type `H.` (note the "`.`") and then press the tab key. You will get a list of available functions (you may need to scroll down to see the whole list). As before, *experiment and explore*—it is really hard to break anything.

Here is another couple of ways to experiment and explore. Find a function that looks interesting, say `is_abelian()`. Type `H.is_abelian?` (note the question mark) followed by the enter key. This will display a portion of the source code for the `is_abelian()` function, describing the inputs and output, possibly illustrated with example uses.

If you want to learn more about how Sage works, or possibly extend its functionality, then you can start by examining the complete Python source code. For example, try `H.is_abelian??`, which will allow you to determine that the `is_abelian()` function is basically riding on GAP's `IsAbelian()` command and asking GAP do the heavy-lifting for us. (To get the maximum advantage of using Sage it helps to know some basic Python programming, but it is not required.)

OK, on to some popular command for groups. If you are using the worksheet, be sure you have defined the group H as the dihedral group D_6, since we will not keep repeating its definition below.

Abelian?

The command

```
sage: H = DihedralGroup(6)
sage: H.is_abelian()
False
```

will return `False` since D_6 is a non-abelian group.

Order

The command

```
sage: H = DihedralGroup(6)
sage: H.order()
12
```

will return `12` since D_6 is a group of with 12 elements.

All elements

The command

```
sage: H = DihedralGroup(6)
sage: H.list()
[(), (2,6)(3,5), (1,2)(3,6)(4,5), (1,2,3,4,5,6), (1,3)(4,6), (1,3,5)(2,4,6), (1,4)(2,3)(5,6), (1,4)(2
```

will return all of the elements of H in a fixed order as a Python list. Indexing (`[ ]`) can be used to extract the individual elements of the list, remembering that counting the elements of the list begins at zero.

```
sage: H = DihedralGroup(6)
sage: elements = H.list()
sage: elements[3]
(1,2,3,4,5,6)
```

Cayley table

The command

```
sage: H = DihedralGroup(6)
sage: H.cayley_table()
*  a b c d e f g h i j k l
 +-----------------------
a| a b c d e f g h i j k l
b| b a d c f e h g j i l k
c| c k a e d g f i h l b j
d| d l b f c h e j g k a i
e| e j k g a i d l f b c h
f| f i l h b j c k e a d g
g| g h j i k l a b d c e f
h| h g i j l k b a c d f e
i| i f h l j b k c a e g d
j| j e g k i a l d b f h c
k| k c e a g d i f l h j b
l| l d f b h c j e k g i a
```

will construct the Cayley table (or "multiplication table") of H. By default the table uses lowercase Latin letters to name the elements of the group. The actual elements used can be found using the `row_keys()` or `column_keys()` commands for the table. For example to determine the fifth element in the table, the element named `e`:

```
sage: H = DihedralGroup(6)
sage: T = H.cayley_table()
sage: headings = T.row_keys()
sage: headings[4]
(1,3)(4,6)
```

Center

The command `H.center()` will return a subgroup that is the center of the group H (see Exercise 2.46 in Judson). Try

```
sage: H = DihedralGroup(6)
sage: H.center().list()
[(), (1,4)(2,5)(3,6)]
```

to see which elements of H commute with *every* element of H.

Cayley graph

For fun, try `show(H.cayley_graph())`.

Subgroups

Cyclic subgroups

If `G` is a group and `a` is an element of the group (try `a = G.random_element()`), then

```
a = G.random_element()
H = G.subgroup([a])
```

will create `H` as the cyclic subgroup of `G` with generator `a`.

For example the code below will:

1. create `G` as the symmetric group on five symbols;
2. specify `sigma` as an element of `G`;
3. use `sigma` as the generator of a cyclic subgroup `H`;
4. list all the elements of `H`.

In more mathematical notation, we might write $\langle (1\,2\,3)(4\,5) \rangle = H \subseteq G = S_5$.

```
sage: G = SymmetricGroup(5)
sage: sigma = G("(1,2,3) (4,5)")
sage: H = G.subgroup([sigma])
sage: H.list()
[(), (4,5), (1,2,3), (1,2,3)(4,5), (1,3,2), (1,3,2)(4,5)]
```

Experiment by trying different permutations for `sigma` and observing the effect on `H`.

Cyclic groups

Groups that are cyclic themselves are both important and rich in structure. The command `CyclicPermutationGroup(n)` will create a permutation group that is cyclic with `n` elements. Consider the following example (note that the indentation of the third line is critical) which will list the elements of a cyclic group of order 20, preceded by the order of each element.

```
sage: n = 20
sage: CN = CyclicPermutationGroup(n)
sage: for g in CN:
...       print g.order(), "  ", g
...
1    ()
20    (1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
10    (1,3,5,7,9,11,13,15,17,19)(2,4,6,8,10,12,14,16,18,20)
20    (1,4,7,10,13,16,19,2,5,8,11,14,17,20,3,6,9,12,15,18)
5    (1,5,9,13,17)(2,6,10,14,18)(3,7,11,15,19)(4,8,12,16,20)
4    (1,6,11,16)(2,7,12,17)(3,8,13,18)(4,9,14,19)(5,10,15,20)
10    (1,7,13,19,5,11,17,3,9,15)(2,8,14,20,6,12,18,4,10,16)
20    (1,8,15,2,9,16,3,10,17,4,11,18,5,12,19,6,13,20,7,14)
5    (1,9,17,5,13)(2,10,18,6,14)(3,11,19,7,15)(4,12,20,8,16)
20    (1,10,19,8,17,6,15,4,13,2,11,20,9,18,7,16,5,14,3,12)
2    (1,11)(2,12)(3,13)(4,14)(5,15)(6,16)(7,17)(8,18)(9,19)(10,20)
20    (1,12,3,14,5,16,7,18,9,20,11,2,13,4,15,6,17,8,19,10)
5    (1,13,5,17,9)(2,14,6,18,10)(3,15,7,19,11)(4,16,8,20,12)
20    (1,14,7,20,13,6,19,12,5,18,11,4,17,10,3,16,9,2,15,8)
10    (1,15,9,3,17,11,5,19,13,7)(2,16,10,4,18,12,6,20,14,8)
4    (1,16,11,6)(2,17,12,7)(3,18,13,8)(4,19,14,9)(5,20,15,10)
5    (1,17,13,9,5)(2,18,14,10,6)(3,19,15,11,7)(4,20,16,12,8)
20    (1,18,15,12,9,6,3,20,17,14,11,8,5,2,19,16,13,10,7,4)
10    (1,19,17,15,13,11,9,7,5,3)(2,20,18,16,14,12,10,8,6,4)
20    (1,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2)
```

By varying the size of the group (change the value of `n`) you can begin to illustrate some of the structure of a cyclic group (for example, try a prime).

We can cut/paste an element of order 5 from the output above (in the case when the cyclic group has 20 elements) and quickly build a subgroup:

```
sage: C20 = CyclicPermutationGroup(20)
sage: rho = C20("(1,17,13,9,5)(2,18,14,10,6)(3,19,15,11,7)(4,20,16,12,8)")
sage: H = C20.subgroup([rho])
sage: H.list()
[(), (1,5,9,13,17)(2,6,10,14,18)(3,7,11,15,19)(4,8,12,16,20), (1,9,17,5,13)(2,10,18,6,14)(3,11,19,7,1
```

For a cyclic group, the following command will list *all* of the subgroups.

```
sage: C20 = CyclicPermutationGroup(20)
sage: C20.conjugacy_classes_subgroups()
[Subgroup of (Cyclic group of order 20 as a permutation group) generated by [()], Subgroup of (Cyclic
```

Be careful, this command uses some more advanced ideas and will not usually list *all* of the subgroups of a group. Here we are relying on special properties of cyclic groups (but see the next section).

If you are viewing this as a PDF, you can safely skip over the next bit of code. However, if you are viewing this as a worksheet in Sage, then this is a place where you can experiment with the structure of the subgroups of a cyclic group. In the input box, enter the order of a cyclic group (numbers between 1 and 40 are good initial choices) and Sage will list each subgroup as a cyclic group with its generator. The factorization at the bottom might help you formulate a conjecture.

```
%auto
@interact
def _(n = input_box(default=12, label = "Cyclic group of order:", type=Integer) ):
    cyclic = CyclicPermutationGroup(n)
    subgroups = cyclic.conjugacy_classes_subgroups()
    html( "All subgroups of a cyclic group of order $%s$\n" % latex(n) )
    table = "$\\begin{array}{ll}"
    for sg in subgroups:
      table = table + latex(sg.order()) + \
              " & \\left\\langle" + latex(sg.gens()[0]) + \
              "\\right\\rangle\\\\"
    table = table + "\\end{array}$"
    html(table)
    html("\nHint: $%s$ factors as $%s$" % ( latex(n), latex(factor(n)) ) )
```

All subgroups

If H is a subgroup of G and $g \in G$, then $gHg^{-1} = \{ghg^{-1} \mid h \in G\}$ will also be a subgroup of G. If `G` is a group, then the command `G.conjugacy_classes_subgroups()` will return a list of subgroups of `G`, but not all of the subgroups. However, every subgroup can be constructed from one on the list by the gHg^{-1} construction with a suitable g. As an illustration, the code below:

1. creates `K` as the dihedral group of order 24, D_{12};
2. stores the list of subgroups output by `K.conjugacy_classes_subgroups()` in the variable `sg`;
3. prints the elements of the list;
4. selects the second subgroup in the list, and lists its elements.

```
sage: K = DihedralGroup(12)
sage: sg = K.conjugacy_classes_subgroups()
sage: print "sg:\n", sg
sg:
[Subgroup of (Dihedral group of order 24 as a permutation group) generated by [()], Subgroup of (Dihe
sage: print "\nAn order two subgroup:\n", sg[1].list()

An order two subgroup:
[(), (1,2)(3,12)(4,11)(5,10)(6,9)(7,8)]
```

It is important to note that this is a nice long list of subgroups, but will rarely create *every* such subgroup. For example, the code below:

1. creates `rho` as an element of the group `K`;
2. creates `L` as a cyclic subgroup of `K`;
3. prints the two elements of `L`; and finally
4. tests to see if this subgroup is part of the output of the list `sg` created just above (it is not).

```
sage: K = DihedralGroup(12)
sage: sg = K.conjugacy_classes_subgroups()
sage: rho = K("(1,4) (2,3) (5,12) (6,11) (7,10) (8,9)")
sage: L = PermutationGroup([rho])
sage: L.list()
[(), (1,4)(2,3)(5,12)(6,11)(7,10)(8,9)]
sage: L in sg
False
```

Symmetry groups

You can give Sage a short list of elements of a permutation group and Sage will find the smallest subgroup that contains those elements. We say the list "generates" the subgroup. We list a few interesting subgroups you can create this way.

Symmetries of an equilateral triangle

Label the vertices of an equilateral triangle as 1, 2 and 3. Then *any* permutation of the vertices will be a symmetry of the triangle. So either `SymmetricGroup(3)` or `DihedralGroup(3)` will create the full symmetry group.

Symmetries of an n-gon

A regular, n-sided figure in the plane (an n-gon) has $2n$ symmetries, comprised of n rotations (including the trivial one) and n "flips" about various axes. The dihedral group `DihedralGroup(n)` is frequently defined as exactly the symmetry group of an n-gon.

Symmetries of a tetrahedron

Label the 4 vertices of a regular tetrahedron as 1, 2, 3 and 4. Fix the vertex labeled 4 and rotate the opposite face through 120 degrees. This will create the permutation/symmetry $(1\,2\,3)$. Similarly, fixing vertex 1, and rotating the opposite face will create the permutation $(2\,3\,4)$. These two permutations are enough to generate the full group of the twelve symmetries of the tetrahedron. Another symmetry can be visualized by running an axis through the midpoint of an edge of the tetrahedron through to the midpoint of the opposite edge, and then rotating by 180 degrees about this

axis. For example, the 1–2 edge is opposite the 3–4 edge, and the symmetry is described by the permutation $(1\,2)(3\,4)$. This permutation, along with either of the above permutations will also generate the group. So here are two ways to create this group:

```
sage: tetra_one = PermutationGroup(["(1,2,3)", "(2,3,4)"])
sage: tetra_one
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: tetra_two = PermutationGroup(["(1,2,3)", "(1,2)(3,4)"])
sage: tetra_two
Permutation Group with generators [(1,2)(3,4), (1,2,3)]
```

This group has a variety of interesting properties, so it is worth experimenting with. You may also know it as the "alternating group on 4 symbols," which Sage will create with the command `AlternatingGroup(4)`.

Symmetries of a cube

Label vertices of one face of a cube with 1, 2, 3 and 4, and on the opposite face label the vertices 5, 6, 7 and 8 (5 opposite 1, 6 opposite 2, etc.). Consider three axes that run from the center of a face to the center of the opposite face, and consider a quarter-turn rotation about each axis. These three rotations will construct the entire symmetry group. Use

```
sage: cube = PermutationGroup(["(3,2,6,7)(4,1,5,8)",
...   "(1,2,6,5)(4,3,7,8)", "(1,2,3,4)(5,6,7,8)"])
sage: cube.list()
[(), (2,4,5)(3,8,6), (2,5,4)(3,6,8), (1,2)(3,5)(4,6)(7,8), (1,2,3,4)(5,6,7,8), (1,2,6,5)(3,7,8,4), (1
```

A cube has four distinct diagonals (joining opposite vertices through the center of the cube). Each symmetry of the cube will cause the diagonals to arrange differently. In this way, we can view an element of the symmetry group as a permutation of four "symbols"—the diagonals. It happens that *each* of the 24 permutations of the diagonals is created by exactly one symmetry of the 8 vertices of the cube. So this subgroup of S_8 is "the same as" S_4. In Sage:

```
sage: cube = PermutationGroup(["(3,2,6,7)(4,1,5,8)",
...   "(1,2,6,5)(4,3,7,8)", "(1,2,3,4)(5,6,7,8)"])
sage: cube.is_isomorphic(SymmetricGroup(4))
True
```

will test to see if the group of symmetries of the cube are "the same as" S_4 and so will return `True`.

Here is an another way to create the symmetries of a cube. Number the six *faces* of the cube as follows: 1 on top, 2 on the bottom, 3 in front, 4 on the right, 5 in back, 6 on the left. Now the same rotations as before (quarter-turns about axes through the centers of two opposite faces) can be used as generators of the symmetry group:

```
sage: cubeface = PermutationGroup(["(1,3,2,5)", "(1,4,2,6)", "(3,4,5,6)"])
sage: cubeface.list()
[(), (3,4,5,6), (3,5)(4,6), (3,6,5,4), (1,2)(4,6), (1,2)(3,4)(5,6), (1,2)(3,5), (1,2)(3,6)(4,5), (1,3
```

Again, this subgroup of S_6 is "same as" the full symmetric group, S_4:

```
sage: cubeface = PermutationGroup(["(1,3,2,5)", "(1,4,2,6)", "(3,4,5,6)"])
sage: cubeface.is_isomorphic(SymmetricGroup(4))
True
```

It turns out that in each of the above constructions, it is sufficient to use just two of the three generators (any two). But one generator is not enough. Give it a try and use Sage to convince yourself that a generator can be sacrificed in each case.

Normal subgroups

Checking normality

The code below:

1. begins with the alternating group A_4;
2. specifies three elements of the group (the three symmetries of the tetrahedron that are 180 degree rotations about axes through midpoints of opposite edges);
3. uses these three elements to generate a subgroup; and finally
4. illustrates the command for testing if the subgroup `H` is a normal subgroup of the group `A4`.

```
sage: A4 = AlternatingGroup(4)
sage: r1 = A4("(1,2)(3,4)")
sage: r2 = A4("(1,3)(2,4)")
sage: r3 = A4("(1,4)(2,3)")
sage: H = A4.subgroup([r1, r2, r3])
sage: H.is_normal(A4)
True
```

Quotient group

Extending the previous example, we can create the quotient (factor) group of A_4 by H. The commands

```
sage: A4 = AlternatingGroup(4)
sage: r1 = A4("(1,2)(3,4)")
sage: r2 = A4("(1,3)(2,4)")
sage: r3 = A4("(1,4)(2,3)")
sage: H = A4.subgroup([r1, r2, r3])
sage: A4.quotient(H)
Permutation Group with generators [(1,2,3)]
```

returns a permutation group generated by `(1,2,3)`. As expected this is a group of order 3. Notice that we do not get back a group of the actual cosets, but instead we get a group *isomorphic* to the factor group.

Simple groups

It is easy to check to see if a group is void of any normal subgroups. The commands

```
sage: AlternatingGroup(5).is_simple()
True
sage: AlternatingGroup(4).is_simple()
False
```

prints `True` and then `False`.

Composition series

For any group, it is easy to obtain a composition series. There is an element of randomness in the algorithm, so you may not always get the same results. (But the list of factor groups is unique, according to the Jordan-Hölder theorem.) Also, the subgroups generated sometimes have more generators than necessary, so you might want to "study" each subgroup carefully by checking properties like its order.

An interesting example is:

```
DihedralGroup(105).composition_series()
```

The output will be a list of 5 subgroups of D_{105}, each a normal subgroup of its predecessor.

Several other series are possible, such as the derived series. Use tab-completion to see the possibilities.

Conjugacy

Given a group G, we can define a relation $\sim$ on G by: for $a, b \in G$, $a \sim b$ if and only if there exists an element $g \in G$ such that $gag^{-1} = b$.

Since this is an equivalence relation, there is an associated partition of the elements of G into equivalence classes. For this very important relation, the classes are known as "conjugacy classes." A representative of each of these equivalence classes can be found as follows. Suppose `G` is a permutation group, then `G.conjugacy_classes_representatives()` will return a list of elements of G, one per conjugacy class.

Given an element $g \in G$, the "centralizer" of g is the set $C(g) = \{h \in G \mid hgh^{-1} = g\}$, which is a subgroup of G. A theorem tells us that the size of each conjugacy class is the order of the group divided by the order of the centralizer of an element of the class. With the following code we can determine the size of the conjugacy classes of the full symmetric group on 5 symbols:

```
sage: G = SymmetricGroup(5)
sage: group_order = G.order()
sage: reps = G.conjugacy_classes_representatives()
sage: class_sizes = []
sage: for g in reps:
...       class_sizes.append(group_order / G.centralizer(g).order())
...
sage: class_sizes
[1, 10, 15, 20, 20, 30, 24]
```

This should produce the list `[1, 10, 15, 20, 20, 30, 24]` which you can check sums to 120, the order of the group. You might be able to produce this list by counting elements of the group S_5 with identical cycle structure (which will require a few simple combinatorial arguments).

Sylow subgroups

Sylow's Theorems assert the existence of certain subgroups. For example, if p is a prime, and p^r divides the order of a group G, then G must have a subgroup of order p^r. Such a subgroup could be found among the output of the `conjugacy_classes_subgroups()` command by checking the orders of the subgroups produced. The `map()` command is a quick way to do this. The symmetric group on 7 symbols, S_7, has order $7! = 5040$ and is divisible by $2^4 = 16$. Let's find one example of a subgroup of permutations on 4 symbols with order 16:

```
sage: G = SymmetricGroup(7)
sage: subgroups = G.conjugacy_classes_subgroups()
sage: map(order, subgroups)
[1, 2, 2, 2, 3, 3, 4, 4, 4, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6, 6, 7, 8, 8, 8, 8, 8, 8, 8, 9, 10, 10,
```

The `map(order, subgroups)` command will apply the `order()` function to each of the subgroups in the list `subgroups`. The output is thus a large list of the orders of many subgroups (96 to be precise). If you count carefully, you will see that the 49-th subgroup has order 16. You can retrieve this group for further study by referencing it as `subgroups[48]` (remember that counting starts at zero).

If p^r is the highest power of p to divide the order of G, then a subgroup of order p^r is known as a "Sylow p-subgroup." Sylow's Theorems also say any two Sylow p-subgroups are conjugate, so the output of

`conjugacy_classes_subgroups()` should only contain each Sylow p-subgroup once. But there is an easier way, `sylow_subgroup(p)` will return one. Notice that the argument of the command is just the prime p, not the full power p^r. Failure to use a prime will generate an informative error message.

Groups of small order as permutation groups

We list here constructions, as permutation groups, for all of the groups of order less than 16.

```
-----------------------------------------------------------------------------------------
Size  Construction                                Notes
-----------------------------------------------------------------------------------------
1     SymmetricGroup(1)                           Trivial
2     SymmetricGroup(2)                           Also CyclicPermutationGroup(2)
3     CyclicPermutationGroup(3)                   Prime order
4     CyclicPermutationGroup(4)                   Cyclic
4     KleinFourGroup()                            Abelian, non-cyclic
5     CyclicPermutationGroup(5)                   Prime order
6     CyclicPermutationGroup(6)                   Cyclic
6     SymmetricGroup(3)                           Non-abelian, also DihedralGroup(3)
7     CyclicPermutationGroup(7)                   Prime order
8     CyclicPermutationGroup(8)                   Cyclic
8     D1 = CyclicPermutationGroup(4)
      D2 = CyclicPermutationGroup(2)
      G = direct_product_permgroups([D1,D2])      Abelian, non-cyclic
8     D1 = CyclicPermutationGroup(2)
      D2 = CyclicPermutationGroup(2)
      D3 = CyclicPermutationGroup(2)
      G = direct_product_permgroups([D1,D2,D3])}  Abelian, non-cyclic
8     DihedralGroup(4)                            Non-abelian
8     QuaternionGroup()}                          Quaternions, also DiCyclicGroup(2)
9     CyclicPermutationGroup(9)                   Cyclic
9     D1 = CyclicPermutationGroup(3)
      D2 = CyclicPermutationGroup(3)
      G = direct_product_permgroups([D1,D2])      Abelian, non-cyclic
10    CyclicPermutationGroup(10)                  Cyclic
10    DihedralGroup(5)                            Non-abelian
11    CyclicPermutationGroup(11)                  Prime order
12    CyclicPermutationGroup(12)                  Cyclic
12    D1 = CyclicPermutationGroup(6)
      D2 = CyclicPermutationGroup(2)
      G = direct_product_permgroups([D1,D2])      Abelian, non-cyclic
12    DihedralGroup(6)                            Non-abelian
12    AlternatingGroup(4)                         Non-abelian, symmetries of tetrahedron
12    DiCyclicGroup(3)                            Non-abelian
                                                  Also semi-direct  product $Z_3 \rtimes Z_4$
13    CyclicPermutationGroup(13)                  Prime order
14    CyclicPermutationGroup(14)                  Cyclic
14    DihedralGroup(7)                            Non-abelian
15    CyclicPermutationGroup(15)                  Cyclic
-----------------------------------------------------------------------------------------
```

Acknowledgements

The construction of Sage is the work of many people, and the group theory portion is made possible by the extensive work of the creators of GAP. However, we will single out three people from the Sage team to thank for major contributions toward bringing you the group theory portion of Sage: David Joyner, William Stein, and Robert Bradshaw.

Thanks!

6.1.5 Lie Methods and Related Combinatorics in Sage

Author: Daniel Bump (Stanford University) and Anne Schilling (UC Davis)

These notes explain how to use the mathematical software Sage for Lie group computations. Sage also contains many combinatorial algorithms. We will cover only some of these.

Preparation of this document was supported in part by NSF grants DMS-0652817, DMS-1001079, OCI-1147463, DMS–0652641, DMS–0652652, DMS–1001256 and and OCI–1147247.

The Scope of this Document

Lie groups and algebras

Sage can be used to do standard computations for Lie groups and Lie algebras. The following categories of representations are equivalent:

- Complex representations of a compact, semisimple simply connected Lie group G.
- Complex representations of its Lie algebra $\mathfrak{g}$. This is a real Lie algebra, so representations are not required to be complex linear maps.
- Complex representations of its complexified Lie algebra $\mathfrak{g}_{\mathbf{C}} = \mathbf{C} \otimes \mathfrak{g}$. This is a complex Lie algebra and representations are required to be complex linear transformations.
- The complex analytic representations of the semisimple simply-connected complex analytic group $G_{\mathbf{C}}$ having $\mathfrak{g}_{\mathbf{C}}$ as its Lie algebra.
- Modules of the universal enveloping algebra $U(\mathfrak{g}_{\mathbf{C}})$.
- Modules of the quantized enveloping algebra $U_q(\mathfrak{g}_{\mathbf{C}})$.

For example, we could take $G = SU(n)$, $\mathfrak{g} = \mathfrak{sl}(n, \mathbf{R})$, $\mathfrak{g}_{\mathbf{C}} = \mathfrak{sl}(n, \mathbf{C})$ and $G_{\mathbf{C}} = SL(n, \mathbf{C})$. Because these categories are the same, their representations may be studied simultaneously. The above equivalences may be expanded to include reductive groups like $U(n)$ and $GL(n)$ with a bit of care.

Here are some typical problems that can be solved using Sage:

- Decompose a module in any one of these categories into irreducibles.
- Compute the Frobenius-Schur indicator of an irreducible module.
- Compute the tensor product of two modules.
- If H is a subgroup of G, study the restriction of modules for G to H. The solution to this problem is called a *branching rule*.
- Find the multiplicities of the weights of the representation.

In addition to its representations, which we may study as above, a Lie group has various related structures. These include:

- The Weyl Group W.
- The Weight Lattice.
- The Root System
- The Cartan Type.

- The Dynkin diagram.
- The extended Dynkin diagram.

Sage contains methods for working with these structures.

If there is something you need that is not implemented, getting it added to Sage will likely be possible. You may write your own algorithm for an unimplemented task, and if it is something others will be interested in, it is probably possible to get it added to Sage.

Combinatorics

Sage supports a great many related mathematical objects. Some of these properly belong to combinatorics. It is beyond the scope of these notes to cover all the combinatorics in Sage, but we will try to touch on those combinatorial methods which have some connection with Lie groups and representation theory. These include:

- The affine Weyl group, an infinite group containing W.
- Kashiwara crystals, which are combinatorial analogs of modules in the above categories.
- Coxeter group methods applicable to Weyl groups and the affine Weyl group, such as Bruhat order.
- The Iwahori Hecke algebras, which are deformations of the group algebras of W and the affine Weyl group.
- Kazhdan-Lusztig polynomials.

Lie Group Basics

Goals of this section

Since we must be brief here, this is not really a place to learn about Lie groups or Lie algebras. Rather, the point of this section is to outline what you need to know to use Sage effectively for Lie computations, and to fix ideas and notations.

Semisimple and reductive groups

If $g \in GL(n, \mathbf{C})$, then g may be uniquely factored as $g_1 g_2$ where g_1 and g_2 commute, with g_1 semisimple (diagonalizable) and g_2 unipotent (all its eigenvalues equal to 1). This follows from the Jordan canonical form. If $g = g_1$ then g is called *semisimple* and if $g = g_2$ then g is called *unipotent*.

We consider a Lie group G and a class of representations such that if an element $g \in G$ is unipotent (resp. semisimple) in one faithful representation from the class, then it is unipotent (resp. semisimple) in every faithful representation of the class. Thus the notion of being semisimple or unipotent is intrinsic. Examples:

- Compact Lie groups with continuous representations
- Complex analytic groups with analytic representations
- Algebraic groups over $\mathbf{R}$ with algebraic representations.

A subgroup of G is called *unipotent* if it is connected and all its elements are unipotent. It is called a *torus* if it is connected, abelian, and all its elements are semisimple. The group G is called *reductive* if it has no nontrivial normal unipotent subgroup. For example, $GL(2, \mathbf{C})$ is reductive, but its subgroup:

$$\left\{ \begin{pmatrix} a & b \\ & d \end{pmatrix} \right\}$$

is not since it has a normal unipotent subgroup

$$\left\{\begin{pmatrix} 1 & b \\ & 1 \end{pmatrix}\right\}.$$

A group has a unique largest normal unipotent subgroup, called the *unipotent radical*, so it is reductive if and only if the unipotent radical is trivial.

A Lie group is called *semisimple* it is reductive and furthermore has no nontrivial normal tori. For example $GL(2, \mathbf{C})$ is reductive but not semisimple because it has a normal torus:

$$\left\{\begin{pmatrix} a & \\ & a \end{pmatrix}\right\}.$$

The group $SL(2, \mathbf{C})$ is semisimple.

Fundamental group and center

If G is a semisimple Lie group then its center and fundamental group are finite abelian groups. The universal covering group $\tilde{G}$ is therefore a finite extension with the same Lie algebra. Any representation of G may be reinterpreted as a representation of the simply connected $\tilde{G}$. Therefore we may as well consider representations of $\tilde{G}$, and restrict ourselves to the simply connected group.

Parabolic subgroups and Levi subgroups

Let G be a reductive complex analytic group. A maximal solvable subgroup of G is called a *Borel subgroup*. All Borel subgroups are conjugate. Any subgroup P containing a Borel subgroup is called a *parabolic subgroup*. We may write P as a semidirect product of its maximal normal unipotent subgroup or *unipotent radical* P and a reductive subgroup M, which is determined up to conjugacy. The subgroup M is called a *Levi subgroup*.

Example: Let $G = GL_n(\mathbf{C})$ and let $r_1, \ldots, r_k$ be integers whose sum is n. Then we may consider matrices of the form:

$$\begin{pmatrix} g_1 & * & \cdots & * \\ & g_2 & & * \\ & & \ddots & \\ & & & g_r \end{pmatrix}$$

where $g_i \in GL(r_i, \mathbf{C})$. The unipotent radical consists of the subgroup in which all $g_i = I_{r_i}$. The Levi subgroup (determined up to conjugacy) is:

$$M = \left\{\begin{pmatrix} g_1 & & & \\ & g_2 & & \\ & & \ddots & \\ & & & g_r \end{pmatrix}\right\},$$

and is isomorphic to $M = GL(r_1, \mathbf{C}) \times \cdots \times GL(r_k, \mathbf{C})$. Therefore M is a Levi subgroup.

The notion of a Levi subgroup can be extended to compact Lie groups. Thus $U(r_1) \times \cdots \times U(r_k)$ is a Levi subgroup of $U(n)$. However parabolic subgroups do not exist for compact Lie groups.

Cartan types

Semisimple Lie groups are classified by their *Cartan types*. There are both reducible and irreducible Cartan types in Sage. Let us start with the irreducible types. Such a type is implemented in Sage as a pair `['X', r]` where 'X' is one of A, B, C, D, E, F or G and r is a positive integer. If 'X' is 'D' then we must have $r > 1$ and if 'X' is one of the *exceptional types* 'E', 'F' or 'G' then r is limited to only a few possibilities. The exceptional types are:

`['G', 2], ['F', 4], ['E', 6], ['E', 7] or ['E', 8].`

A simply-connected semisimple group is a direct product of simple Lie groups, which are given by the following table. The integer r is called the *rank*, and is the dimension of the maximal torus.

Here are the Lie groups corresponding to the classical types:

compact group	complex analytic group	Cartan type
$SU(r+1)$	$SL(r+1, \mathbf{C})$	A_r
$spin(2r+1)$	$spin(2r+1, \mathbf{C})$	B_r
$Sp(2r)$	$Sp(2r, \mathbf{C})$	C_r
$spin(2r)$	$spin(2r, \mathbf{C})$	D_r

You may create these Cartan types and their Dynkin diagrams as follows:

```
sage: ct = CartanType("D5"); ct
['D', 5]
```

Here `"D5"` is an abbreviation for `['D',5]`. The group $spin(n)$ is the simply-connected double cover of the orthogonal group $SO(n)$.

Dual Cartan types

Every Cartan type has a dual, which you can get from within Sage:

```
sage: CartanType("B4").dual()
['C', 4]
```

Types other than B_r and C_r for $r > 2$ are self-dual in the sense that the dual is isomorphic to the original type; however the isomorphism of a Cartan type with its dual might relabel the vertices. We can see this as follows:

```
sage: CartanType("F4").dynkin_diagram()
O---O=>=O---O
1   2   3   4
F4
sage: CartanType("F4").dual()
['F', 4] relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
sage: CartanType("F4").dual().dynkin_diagram()
O---O=>=O---O
4   3   2   1
F4 relabelled by {1: 4, 2: 3, 3: 2, 4: 1}
```

Reducible Cartan types

If G is a Lie group of finite index in $G_1 \times G_2$, where G_1 and G_2 are Lie groups of positive dimension, then G is called *reducible*. In this case, the root system of G is the disjoint union of the root systems of G_1 and G_2, which lie in orthogonal subspaces of the ambient space of the weight space of G. The Cartan type of G is thus *reducible*.

Reducible Cartan types are supported in Sage as follows:

```
sage: RootSystem("A1xA1")
Root system of type A1xA1
sage: WeylCharacterRing("A1xA1")
The Weyl Character Ring of Type A1xA1 with Integer Ring coefficients
```

Low dimensional Cartan types

There are some isomorphisms that occur in low degree.

Cartan Type	Group	Equivalent Type	Isomorphic Group
B_2	$spin(5)$	C_2	$Sp(4)$
D_3	$spin(6)$	A_3	$SL(4)$
D_2	$spin(4)$	$A1 \times A_1$	$SL(2) \times SL(2)$
B_1	$spin(3)$	A_1	$SL(2)$
C_1	$Sp(2)$	A_1	$SL(2)$

Sometimes the redundant Cartan types such as D_3 and D_2 are excluded from the list of Cartan types. However Sage allows them since excluding them leads to exceptions having to be made in algorithms. A better approach, which is followed by Sage, is to allow the redundant Cartan types, but to implement the isomorphisms explicitly as special cases of branching rules. The utility of this approach may be seen by considering that the rank one group $SL(2)$ has different natural weight lattices realizations depending on whether we consider it to be $SL(2)$, $spin(2)$ or $Sp(2)$:

```
sage: RootSystem("A1").ambient_space().simple_roots()
Finite family {1: (1, -1)}
sage: RootSystem("B1").ambient_space().simple_roots()
Finite family {1: (1)}
sage: RootSystem("C1").ambient_space().simple_roots()
Finite family {1: (2)}
```

Relabeled Cartan types

By default Sage uses the labeling of the Dynkin diagram from [Bourbaki46]. There is another labeling of the vertices due to Dynkin. Most of the literature follows [Bourbaki46], though [Kac] follows Dynkin.

If you need to use Dynkin's labeling, you should be aware that Sage does support relabeled Cartan types. See the documentation in `sage.combinat.root_system.type_relabel` for further information.

Standard realizations of the ambient spaces

These realizations follow the Appendix in [Bourbaki46]. See the *Root system plot tutorial* for how to visualize them.

Type A For type A_r we use an $r+1$ dimensional ambient space. This means that we are modeling the Lie group $U(r+1)$ or $GL(r+1, \mathbf{C})$ rather than $SU(r+1)$ or $SL(r+1, \mathbf{C})$. The ambient space is identified with $\mathbf{Q}^{r+1}$:

```
sage: RootSystem("A3").ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1)}
sage: RootSystem("A3").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1, 1, 1, 0)}
sage: RootSystem("A3").ambient_space().rho()
(3, 2, 1, 0)
```

The dominant weights consist of integer $r+1$-tuples $\lambda = (\lambda_1, \ldots, \lambda_{r+1})$ such that $\lambda_1 \geq \cdots \geq \lambda_{r+1}$.

See *SL versus GL* for further remarks about Type A.

Type B For the remaining classical Cartan types B_r, C_r and D_r we use an r-dimensional ambient space:

```
sage: RootSystem("B3").ambient_space().simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 1)}
sage: RootSystem("B3").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
sage: RootSystem("B3").ambient_space().rho()
(5/2, 3/2, 1/2)
```

This is the Cartan type of $spin(2r+1)$. The last fundamental weight `(1/2, 1/2, ..., 1/2)` is the highest weight of the 2^r dimensional *spin representation*. All the other fundamental representations factor through the homomorphism $spin(2r+1) \to SO(2r+1)$ and are representations of the orthogonal group.

The dominant weights consist of r-tuples of integers or half-integers $(\lambda_1, \ldots, \lambda_r)$ such that $\lambda_1 \geq \lambda_2 \cdots \geq \lambda_r \geq 0$, and such that the differences $\lambda_i - \lambda_j \in \mathbf{Z}$.

Type C

```
sage: RootSystem("C3").ambient_space().simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 2)}
sage: RootSystem("C3").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1, 1, 1)}
sage: RootSystem("C3").ambient_space().rho()
(3, 2, 1)
```

This is the Cartan type of the symplectic group $Sp(2r)$.

The dominant weights consist of r-tuples of integers $\lambda = (\lambda_1, \ldots, \lambda_{r+1})$ such that $\lambda_1 \geq \cdots \geq \lambda_r \geq 0$.

Type D

```
sage: RootSystem("D4").ambient_space().simple_roots()
Finite family {1: (1, -1, 0, 0), 2: (0, 1, -1, 0), 3: (0, 0, 1, -1), 4: (0, 0, 1, 1)}
sage: RootSystem("D4").ambient_space().fundamental_weights()
Finite family {1: (1, 0, 0, 0), 2: (1, 1, 0, 0), 3: (1/2, 1/2, 1/2, -1/2), 4: (1/2, 1/2, 1/2, 1/2)}
sage: RootSystem("D4").ambient_space().rho()
(3, 2, 1, 0)
```

This is the Cartan type of $spin(2r)$. The last two fundamental weights are the highest weights of the two 2^{r-1}-dimensional spin representations.

The dominant weights consist of r-tuples of integers $\lambda = (\lambda_1, \ldots, \lambda_{r+1})$ such that $\lambda_1 \geq \cdots \geq \lambda_{r-1} \geq |\lambda_r|$.

Exceptional Types We leave the reader to examine the exceptional types. You can use Sage to list the fundamental dominant weights and simple roots.

Weights and the ambient space

Let G be a reductive complex analytic group. Let T be a maximal torus, $\Lambda = X^*(T)$ be its group of analytic characters. Then $T \cong (\mathbf{C}^\times)^r$ for some r and $\Lambda \cong \mathbf{Z}^r$.

Example 1: Let $G = \mathrm{GL}_{r+1}(\mathbf{C})$. Then T is the diagonal subgroup and $X^*(T) \cong \mathbf{Z}^{r+1}$. If $\lambda = (\lambda_1, \ldots, \lambda_n)$ then λ is identified with the rational character

$$\mathbf{t} = \begin{pmatrix} t_1 & & \\ & \ddots & \\ & & t_n \end{pmatrix} \longmapsto \prod t_i^{\lambda_i}.$$

Example 2: Let $G = \mathrm{SL}_{r+1}(\mathbf{C})$. Again T is the diagonal subgroup but now if $\lambda \in \mathbf{Z}^{\Delta} = \{(d, \cdots, d) | d \in \mathbf{Z}\} \subseteq \mathbf{Z}^{r+1}$ then $\prod t_i^{\lambda_i} = \det(\mathbf{t})^d = 1$, so $X^*(T) \cong \mathbf{Z}^{r+1}/\mathbf{Z}^{\Delta} \cong \mathbf{Z}^r$.

- Elements of Λ are called *weights*.
- If $\pi : G \to GL(V)$ is any representation we may restrict π to T. Then the characters of T that occur in this restriction are called the *weights of* π.
- G acts on its Lie algebra by conjugation (the *adjoint representation*).
- The nonzero weights of the adjoint representation are called *roots*.
- The *ambient space* of Λ is $\mathbf{Q} \otimes \Lambda$.

The root system

As we have mentioned, G acts on its complexified Lie algebra $\mathfrak{g}_{\mathbf{C}}$ by the adjoint representation. The zero weight space $\mathfrak{g}_{\mathbf{C}}(0)$ is just the Lie algebra of T itself. The other nonzero weights each appear with multiplicity one and form an interesting configuration of vectors called the *root system* Φ.

It is convenient to partition Φ into two sets Φ^+ and Φ^- such that Φ^+ consists of all roots lying on one side of a hyperplane. Often we arrange things so that G is embedded in $GL(n, \mathbf{C})$ in such a way that the positive weights correspond to upper triangular matrices. Thus if α is a positive root, its weight space $\mathfrak{g}_{\mathbf{C}}(\alpha)$ is spanned by a vector X_α, and the exponential of this eigenspace in G is a one-parameter subgroup of unipotent matrices. It is always possible to arrange that this one-parameter subgroup consists of upper triangular matrices.

If α is a positive root that cannot be decomposed as a sum of other positive roots, then α is called a *simple root*. If G is semisimple of rank r, then r is the number of positive roots. Let $\alpha_1, \ldots, \alpha_r$ be these.

The Weyl group

Let G be a complex analytic group. Let T be a maximal torus, and let $N(T)$ be its normalizer. Let $W = N(T)/T$ be the *Weyl group*. It acts on T by conjugation; therefore it acts on the weight lattice Λ and its ambient space. The ambient space admits an inner product that is invariant under this action. Let $(v|w)$ denote this inner product. If α is a root let r_α denote the reflection in the hyperplane of the ambient space that is perpendicular to α. If $\alpha = \alpha_i$ is a simple root, then we use the notation s_i to denote r_α.

Then $s_1, \ldots, s_r$ generate W, which is a *Coxeter group*. This means that it is generated by elements s_i of order two and that if $m(i, j)$ is the order of $s_i s_j$, then

$$W = \left\langle s_i \mid s_i^2 = 1, (s_i s_j)^{m(i,j)} = 1 \right\rangle$$

is a presentation. An important function $\ell : W \to \mathbf{Z}$ is the *length* function, where $\ell(w)$ is the length of the shortest decomposition of w into a product of simple reflections.

The dual root system

The *coroots* are certain linear functionals on the ambient space that also form a root system. Since the ambient space admits a W-invariant inner product $(\ |\ )$, they may be identified with elements of the ambient space itself. Then they

are proportional to the roots, though if the roots have different lengths, long roots correspond to short coroots and conversely. The coroot corresponding to the root α is

$$\alpha^\vee = \frac{2\alpha}{(\alpha|\alpha)}.$$

We can also describe the natural pairing between coroots and roots using this invariant inner product as

$$\langle \alpha^\vee, \beta \rangle = 2\frac{(\alpha|\beta)}{(\alpha|\alpha)}.$$

The Dynkin diagram

The Dynkin diagram is a graph whose vertices are in bijection with the set simple roots. We connect the vertices corresponding to roots that are not orthogonal. Usually two such roots (vertices) make an angle of $2\pi/3$, in which case we connect them with a single bond. Occasionally they may make an angle of $3\pi/4$ in which case we connect them with a double bond, or $5\pi/6$ in which case we connect them with a triple bond. If the bond is single, the roots have the same length with respect to the inner product on the ambient space. In the case of a double or triple bond, the two simple roots in questions have different length, and the bond is drawn as an arrow from the long root to the short root. Only the exceptional group G_2 has a triple bond.

There are various ways to get the Dynkin diagram in Sage:

```
sage: DynkinDiagram("D5")
        O 5
        |
        |
O---O---O---O
1   2   3   4
D5
sage: ct = CartanType("E6"); ct
['E', 6]
sage: ct.dynkin_diagram()
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6
sage: B4 = WeylCharacterRing("B4"); B4
The Weyl Character Ring of Type B4 with Integer Ring coefficients
sage: B4.dynkin_diagram()
O---O---O=>=O
1   2   3   4
B4
sage: RootSystem("G2").dynkin_diagram()
  3
O=<=O
1   2
G2
```

The Cartan matrix

Consider the natural pairing $\langle\ ,\ \rangle$ between coroots and roots, then the defining matrix of this pairing is called the *Cartan matrix*. That is to say, the Cartan matrix $A = (a_{ij})_{ij}$ is given by

$$a_{ij} = \langle \alpha_i^\vee, \alpha_j \rangle.$$

This uniquely corresponds to a root system/Dynkin diagram/Lie group.

We note that we have made a convention choice, and the opposite convention corresponds to taking the transpose of the Cartan matrix.

Fundamental weights and the Weyl vector

There are certain weights $\omega_1, \ldots, \omega_r$ that:

$$\langle \omega_j, \alpha_i \rangle = 2\frac{(\alpha_i|\omega_j)}{(\alpha_i|\alpha_i)} = \delta_{ij}.$$

If G is semisimple then these are uniquely determined, whereas if G is reductive but not semisimple we may choose them conveniently.

Let ρ be the sum of the fundamental dominant weights. If G is semisimple, then ρ is half the sum of the positive roots. In case G is not semisimple, we have noted, the fundamental weights are not completely determined by the inner product condition given above. If we make a different choice, then ρ is altered by a vector that is orthogonal to all roots. This is a harmless change for many purposes such as the Weyl character formula.

In Sage, this issue arises only for Cartan type A_r. See *SL versus GL*.

Representations and characters

Let T be a maximal torus and $\Lambda = X^*(T)$ be the group of rational characters. Then $\Lambda \cong \mathbf{Z}^r$.

- Recall that elements of $\Lambda \cong \mathbf{Z}^r$ are called *weights*.
- The Weyl group $W = N(T)/T$ acts on T, hence on Λ and its ambient space by conjugation.
- The ambient space $\mathbf{Q} \otimes X^*(T) \cong \mathbf{Q}^r$ has a fundamental domain $\mathcal{C}^+$ for the Weyl group W called the *positive Weyl chamber*. Weights in $\mathcal{C}^+$ are called *dominant*.
- Then $\mathcal{C}^+$ consists of all vectors such that $(\alpha|v) \geq 0$ for all positive roots α.
- It is useful to embed Λ in $\mathbf{R}^r$ and consider weights as lattice points.
- If (π, V) is a representation then restricting to T, the module V decomposes into a direct sum of weight eigenspaces $V(\mu)$ with multiplicity $m(\mu)$ for weight μ.
- There is a unique *highest weight* λ with respect to the partial order. We have $\lambda \in \mathcal{C}$ and $m(\lambda) = 1$.
- $V \longleftrightarrow \lambda$ gives a bijection between irreducible representations and weights λ in $\mathcal{C}^+$.

Assuming that G is simply-connected (or more generally, reductive with a simply-connected derived group) every dominant weight λ is the highest weight of a unique irreducible representation π_λ, and $\lambda \mapsto \pi_\lambda$ gives a parametrization of the isomorphism classes of irreducible representations of G by the dominant weights.

The *character* of π_λ is the function $\chi_\lambda(g) = tr(\pi_\lambda(g))$. It is determined by its values on T. If $(z) \in T$ and $\mu \in \Lambda$, let us write $\mathbf{z}^\mu$ for the value of μ on $\mathbf{z}$. Then the character:

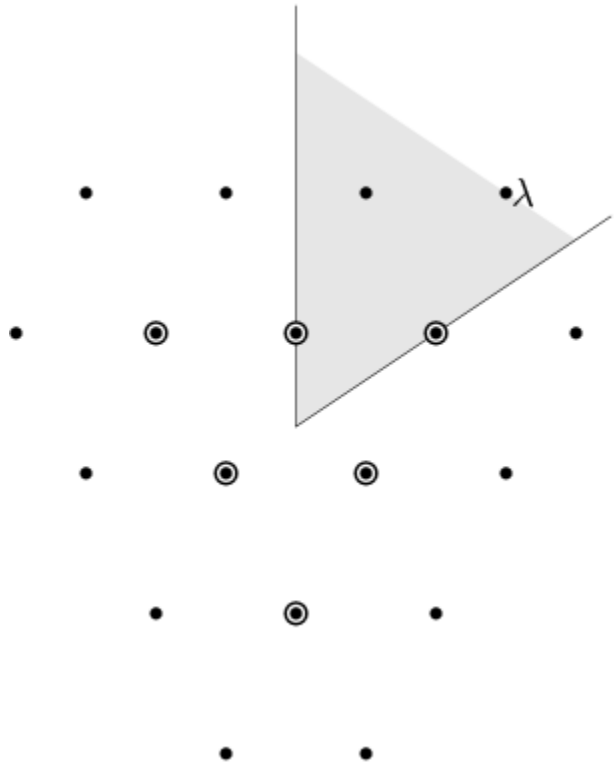
$$\chi_\lambda(\mathbf{z}) = \sum_{\mu \in \Lambda} m(\mu)\, \mathbf{z}^\lambda.$$

Sometimes this is written

$$\chi_\lambda = \sum_{\mu \in \Lambda} m(\mu)\, e^\lambda.$$

The meaning of e^λ is subject to interpretation, but we may regard it as the image of the additive group Λ in its group algebra. The character is then regarded as an element of this ring, the group algebra of Λ.

Representations: an example



In this example, $G = \mathrm{SL}(3, \mathbf{C})$. We have drawn the weights of an irreducible representation with highest weight λ. The shaded region is $\mathcal{C}^+$. λ is a dominant weight, and the labeled vertices are the weights with positive multiplicity in $V(\lambda)$. The weights weights on the outside have $m(\mu) = 1$, while the six interior weights (with double circles) have $m(\mu) = 2$.

Partitions and Schur polynomials

The considerations of this section are particular to type A. We review the relationship between characters of $GL(n, \mathbf{C})$ and symmetric function theory.

A *partition* λ is a sequence of descending nonnegative integers:

$$\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n), \qquad \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0.$$

We do not distinguish between two partitions if they differ only by some trailing zeros, so $(3, 2) = (3, 2, 0)$. If l is the last integer such that $\lambda_l > 0$ then we say that l is the *length* of λ. If $k = \sum \lambda_i$ then we say that λ is a *partition* of k and write $\lambda \vdash k$.

A partition of length $\leq n = r + 1$ is therefore a dominant weight of type `['A',r]`. Not every dominant weight is a partition, since the coefficients in a dominant weight could be negative. Let us say that an element $\mu = (\mu_1, \mu_2, \cdots, \mu_n)$ of the `['A',r]` root lattice is *effective* if the $\mu_i \geq 0$. Thus an effective dominant weight of `['A',r]` is a partition of length $\leq n$, where $n = r + 1$.

Let λ be a dominant weight, and let χ_λ be the character of $GL(n, \mathbf{C})$ with highest weight λ. If k is any integer we may consider the weight $\mu = (\lambda_1 + k, \ldots, \lambda_n + k)$ obtained by adding k to each entry. Then $\chi_\mu = \det{}^k \otimes \chi_\lambda$. Clearly by choosing k large enough, we may make μ effective.

So the characters of irreducible representations of $GL(n, \mathbf{C})$ do not all correspond to partitions, but the characters indexed by partitions (effective dominant weights) are enough that we can write any character $\det{}^{-k} \chi_\mu$ where μ is a

partition. If we take $k = -\lambda_n$ we could also arrange that the last entry in λ is zero.

If λ is an effective dominant weight, then every weight that appears in χ_λ is effective. (Indeed, it lies in the convex hull of $w(\lambda)$ where w runs through the Weyl group $W = S_n$.) This means that if

$$g = \begin{pmatrix} z_1 & & \\ & \ddots & \\ & & z_n \end{pmatrix} \in GL(n, \mathbf{C})$$

then $\chi_\lambda(g)$ is a polynomial in the eigenvalues of g. This is the *Schur polynomial* $s_\lambda(z_1, \ldots, z_n)$.

Affine Cartan types

There are also affine Cartan types, which correspond to (infinite dimensional) affine Lie algebras. There are affine Cartan types of the form `['X', r, 1]` if `X=A,B,C,D,E,F,G` and `['X', r]` is an ordinary Cartan type. There are also *twisted affine types* of the form `[X, r, k]`, where $k = 2$ or 3 if the Dynkin diagram of the ordinary Cartan type `[X, r]` has an automorphism of degree k. When $k = 1$, the affine Cartan type is said to be *untwisted*.

Illustrating some of the methods available for the untwisted affine Cartan type `['A', 4, 1]`:

```
sage: ct = CartanType(['A',4,1]); ct
['A', 4, 1]
sage: ct.dual()
['A', 4, 1]
sage: ct.classical()
['A', 4]
sage: ct.dynkin_diagram()
0
O-----------+
|           |
|           |
O---O---O---O
1   2   3   4
A4~
```

The twisted affine Cartan types are relabeling of the duals of certain untwisted Cartan types:

```
sage: CartanType(['A',3,2])
['B', 2, 1]^*
sage: CartanType(['D',4,3])
['G', 2, 1]^* relabelled by {0: 0, 1: 2, 2: 1}
```

The affine root and the extended Dynkin diagram

For the extended Dynkin diagram, we add one negative root α_0. For the untwisted types, this is the root whose negative is the highest weight in the adjoint representation. Sometimes this is called the *affine root*. We make the Dynkin diagram as before by measuring the angles between the roots. This extended Dynkin diagram is useful for many purposes, such as finding maximal subgroups and for describing the affine Weyl group.

In particular, the hyperplane for the reflection r_0, used in generating the affine Weyl group is translated off the origin (so it becomes an affine hyperplane). Now the root system is not described as linear transformations on an Euclidean space, but instead by *affine* transformations. Thus the dominant chamber has finite volume and tiles the Eucledian space. Moreover, each such tile corresponds to a unique element in the affine Weyl group.

The extended Dynkin diagram may be obtained as the Dynkin diagram of the corresponding untwisted affine type:

```
sage: ct = CartanType("E6"); ct
['E', 6]
sage: ct.affine()
['E', 6, 1]
sage: ct.affine() == CartanType(['E',6,1])
True
sage: ct.affine().dynkin_diagram()
        O 0
        |
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
```

The extended Dynkin diagram is also a method of the `WeylCharacterRing`:

```
sage: WeylCharacterRing("E7").extended_dynkin_diagram()
            O 2
            |
            |
O---O---O---O---O---O---O
0   1   3   4   5   6   7
E7~
```

We note the following important distinctions from the classical cases:

- The affine Weyl groups are all infinte.
- Type $A_1^{(1)}$ has two anti-parallel roots with distinct reflections. The Dynkin diagram in this case is represented by a double bond with arrows going in both directions.

Twisted affine root systems

For the construction of α_0 in the twisted types, we refer the reader to Chaper 8 of [Kac]. As mentioned above, most twisted types can be constructed by the taking the dual root system of an untwisted type. However the type $A_{2n}^{(2)}$ root system which can only be constructed by the twisting procedure defined in [Kac]. It has the following properties:

- The Dynkin diagram of type $A_2^{(2)}$ has a quadruple bond with an arrow pointing from the short root to the long root.
- Type $A_{2n}^{(2)}$ for $n > 1$ has 3 different root lengths.

Further Generalizations

If a root system (on an Euclidean space) has only the angles $\pi/2, 2\pi/3, 3\pi/4, 5\pi/6$ between its roots, then we call the root system *crystallographic* (on Wikipedia article Root_system, this condition is called integrality since for any two roots we have $\langle\beta, \alpha\rangle \in \mathbf{Z}$). So if we look at the reflection group generated by the roots (this is not a Weyl group), we get general Coxeter groups (with non-infinite labels) and non-crystallographic Coxeter groups are not connected with Lie theory.

However we can generalize Dynkin diagrams (equivalently Cartan matrices) to have all its edges labelled by (a, b) where $a, b \in \mathbf{Z}_{>0}$ and corresponds to having a arrows point one way and b arrows pointing the other. For example in type $A_1^{(1)}$, we have one edge of $(2, 2)$, or in type $A_2^{(2)}$, we have one edge of $(1, 4)$ (equivalently $(4, 1)$). These edge

label between i and j corresponds to the entries a_{ij} and a_{ji} in the Cartan matrix. These are used to construct a class of (generally infinite dimensional) Lie algebras called Kac-Moody (Lie) algebras, which in turn are used to construct quantum groups. We refer the reader to [Kac] and [HongKang2002] for more information.

The Weyl Character Ring

Weyl character rings

The Weyl character ring is the representation ring of a compact Lie group. It has a basis consisting of the irreducible representations of G, or equivalently, their characters. The addition and multiplication in the Weyl character ring correspond to direct sum and tensor product of representations.

Methods of the ambient space

In Sage, many useful features of the Lie group are available as methods of the ambient space:

```
sage: S = RootSystem("B2").ambient_space(); S
Ambient space of the Root system of type ['B', 2]
sage: S.roots()
[(1, -1), (1, 1), (1, 0), (0, 1), (-1, 1), (-1, -1), (-1, 0), (0, -1)]
sage: S.fundamental_weights()
Finite family {1: (1, 0), 2: (1/2, 1/2)}
sage: S.positive_roots()
[(1, -1), (1, 1), (1, 0), (0, 1)]
sage: S.weyl_group()
Weyl Group of type ['B', 2] (as a matrix group acting on the ambient space)
```

Methods of the Weyl character ring

If you are going to work with representations, you may want to create a *Weyl Character ring*. Many methods of the ambient space are available as methods of the Weyl character ring:

```
sage: B3 = WeylCharacterRing("B3")
sage: B3.fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0), 3: (1/2, 1/2, 1/2)}
sage: B3.simple_roots()
Finite family {1: (1, -1, 0), 2: (0, 1, -1), 3: (0, 0, 1)}
sage: B3.dynkin_diagram()
O---O=>=O
1   2   3
B3
```

Other useful methods of the Weyl character ring include:

- `cartan_type`
- `highest_root`
- `positive_root`
- `extended_dynkin_diagram`
- `rank`

Some methods of the ambient space are not implemented as methods of the Weyl character ring. However, the ambient space itself is a method, and so you have access to its methods from the Weyl character ring:

```
sage: B3 = WeylCharacterRing("B3")
sage: B3.space().weyl_group()
Weyl Group of type ['B', 3] (as a matrix group acting on the ambient space)
sage: B3.space()
Ambient space of the Root system of type ['B', 3]
sage: B3.space().rho()
(5/2, 3/2, 1/2)
sage: B3.cartan_type()
['B', 3]
```

Coroot notation

It is useful to give the Weyl character ring a name that corresponds to its Cartan type. This has the effect that the ring can parse its own output:

```
sage: G2 = WeylCharacterRing("G2")
sage: [G2(fw) for fw in G2.fundamental_weights()]
[G2(1,0,-1), G2(2,-1,-1)]
sage: G2(1,0,-1)
G2(1,0,-1)
```

Actually the prefix for the ring is configurable, so you don't really have to call this ring `G2`. Type `WeylCharacterRing?` at the `sage:` prompt for details.

There is one important option that you may want to know about. This is *coroot notation*. You select this by specifying the option `style="coroots"` when you create the ring. With the coroot style, the fundamental weights are represented `(1,0,0, ...)`, `(0,1,0,...)` instead of as vectors in the ambient space:

```
sage: B3 = WeylCharacterRing("B3", style="coroots")
sage: [B3(fw) for fw in B3.fundamental_weights()]
[B3(1,0,0), B3(0,1,0), B3(0,0,1)]
sage: B3(0,0,1)
B3(0,0,1)
sage: B3(0,0,1).degree()
8
```

The last representation is the eight dimensional spin representation of $G = spin(7)$, the double cover of the orthogonal group $SO(7)$. In the default notation it would be represented `B3(1/2,1/2,1/2)`.

With the coroot notation every irreducible representation is represented `B3(a,b,c)` where `a`, `b` and `c` are nonnegative integers. This is often convenient. For many purposes the coroot style is preferable.

One disadvantage: in the coroot style the Lie group or Lie algebra is treated as semisimple, so you lose the distinction between $GL(n)$ and $SL(n)$; you also some information about representations of E6 and E7 for the same reason.

Tensor products of representations

The multiplication in the Weyl character ring corresponds to tensor product. This gives us a convenient way of decomposing a tensor product into irreducibles:

```
sage: B3 = WeylCharacterRing("B3")
sage: fw = B3.fundamental_weights()
sage: spinweight = fw[3]; spinweight
```

```
(1/2, 1/2, 1/2)
sage: spin = B3(spinweight); spin
B3(1/2,1/2,1/2)
sage: spin.degree()
8
```

The element $spin$ of the WeylCharacterRing is the representation corresponding to the third highest weight representation, the eight-dimensional spin representation of $spin(7)$. We could just as easily construct it with the commmand:

```
sage: spin = B3(1/2,1/2,1/2)
```

We may compute its tensor product with itself, using the multiplicative structure of the Weyl character ring:

```
sage: chi = spin*spin; chi
B3(0,0,0) + B3(1,0,0) + B3(1,1,0) + B3(1,1,1)
```

We have taken the eight-dimensional spin representation and tensored with itself. We see that the tensor square splits into four irreducibles, each with multiplicity one.

The highest weights that appear here are available (with their coefficients) through the usual free module accessors:

```
sage: from pprint import pprint
sage: list(chi)                              # random
[((1, 1, 1), 1), ((0, 0, 0), 1), ((1, 0, 0), 1), ((1, 1, 0), 1)]
sage: sorted(chi, key=str)
[((0, 0, 0), 1), ((1, 0, 0), 1), ((1, 1, 0), 1), ((1, 1, 1), 1)]
sage: pprint(dict(chi))
{(0, 0, 0): 1, (1, 0, 0): 1, (1, 1, 0): 1, (1, 1, 1): 1}
sage: chi.monomials()
[B3(0,0,0), B3(1,0,0), B3(1,1,0), B3(1,1,1)]
sage: chi.support()
[(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1)]
sage: chi.coefficients()
[1, 1, 1, 1]
sage: [r.degree() for r in chi.monomials()]
[1, 7, 21, 35]
sage: sum(r.degree() for r in chi.monomials())
64
```

Here we have extracted the individual representations, computed their degrees and checked that they sum up to 64.

Weight multiplicities

The weights of the character are available (with their coefficients) through the method `weight_multiplicities`. Continuing from the example in the last section:

```
sage: pprint(chi.weight_multiplicities())
{(0, 0, 0): 8, (-1, 0, 0): 4, (-1, -1, 0): 2, (-1, -1, -1): 1,
 (-1, -1, 1): 1, (-1, 1, 0): 2, (-1, 1, -1): 1, (-1, 1, 1): 1,
 (-1, 0, -1): 2, (-1, 0, 1): 2, (1, 0, 0): 4, (1, -1, 0): 2,
 (1, -1, -1): 1, (1, -1, 1): 1, (1, 1, 0): 2, (1, 1, -1): 1,
 (1, 1, 1): 1, (1, 0, -1): 2, (1, 0, 1): 2, (0, -1, 0): 4,
 (0, -1, -1): 2, (0, -1, 1): 2, (0, 1, 0): 4, (0, 1, -1): 2,
 (0, 1, 1): 2, (0, 0, -1): 4, (0, 0, 1): 4}
```

Each key of this dictionary is a weight, and its value is the multiplicity of that weight in the character.

Example

Suppose that we wish to compute the integral

$$\int_{U(n)} |tr(g)|^{2k}\, dg$$

for various n. Here $U(n)$ is the unitary group, which is the maximal compact subroup of $GL(n,\mathbb{C})$. The irreducible unitary representations of $U(n)$ may be regarded as the basis elements of the WeylCharacterRing of type A_r, where $r = n-1$ so we might work in that ring. The trace $tr(g)$ is then just the character of the standard representation. We may realize it in the WeylCharacterRing by taking the first fundamental weight and coercing it into the ring. For example, if $k = 5$ and $n = 3$ so $r = 2$:

```
sage: A2 = WeylCharacterRing("A2")
sage: fw = A2.fundamental_weights(); fw
Finite family {1: (1, 0, 0), 2: (1, 1, 0)}
sage: tr = A2(fw[1]); tr
A2(1,0,0)
```

We may compute the norm square the character `tr^5` by decomposing it into irreducibles, and taking the sum of the squares of their multiplicities. By Schur orthogonality, this gives the inner product of the $tr(g)^5$ with itself, that is, the integral of $|tr(g)|^{10}$:

```
sage: sum(d^2 for d in (tr^5).coefficients())
103
```

So far we have been working with $n = 3$. For general n:

```
sage: def f(n,k):
....:     R = WeylCharacterRing(['A',n-1])
....:     tr = R(R.fundamental_weights()[1])
....:     return sum(d^2 for d in (tr^k).coefficients())
sage: [f(n,5) for n in [2..7]]
[42, 103, 119, 120, 120, 120]
```

We see that the 10-th moment of $tr(g)$ is just 5! when n is sufficiently large. What if we fix n and vary k?

```
sage: [f(2,k) for k in [1..10]]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796]
sage: [catalan_number(k) for k in [1..10]]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796]
```

Frobenius-Schur indicator

The Frobeinus-Schur indicator of an irreducible representation of a compact Lie group G with character χ is:

$$\int_G \chi(g^2)\, dg$$

The Haar measure is normalized so that $vol(G) = 1$. The Frobenius-Schur indicator equals 1 if the representation is real (orthogonal), -1 if it is quaternionic (symplectic) and 0 if it is complex (not self-contragredient). This is a method of weight ring elements corresponding to irreducible representations. Let us compute the Frobenius-Schur indicators of the spin representations of some odd spin groups:

```
sage: def spinrepn(r):
....:     R = WeylCharacterRing(['B',r])
....:     return R(R.fundamental_weights()[r])
....:
sage: spinrepn(3)
B3(1/2,1/2,1/2)
sage: for r in [1..4]: print r, spinrepn(r).frobenius_schur_indicator()
1 -1
2 -1
3 1
4 1
```

Here we have defined a function that returns the spin representation of the group $spin(2r+1)$ with Cartan type $['B', r]$, then computed the Frobenius-Schur indicators for a few values. From this experiment we see that the spin representations of $spin(3)$ and $spin(5)$ are symplectic, while those of $spin(7)$ and $spin(9)$ are orthogonal.

Symmetric and exterior powers

Sage can compute symmetric and exterior powers of a representation:

```
sage: B3 = WeylCharacterRing("B3",style="coroots")
sage: spin = B3(0,0,1); spin.degree()
8
sage: spin.exterior_power(2)
B3(1,0,0) + B3(0,1,0)
sage: spin.exterior_square()
B3(1,0,0) + B3(0,1,0)
sage: spin.exterior_power(5)
B3(0,0,1) + B3(1,0,1)
sage: spin.symmetric_power(5)
B3(0,0,1) + B3(0,0,3) + B3(0,0,5)
```

The k-th exterior square of a representation is zero if k is greater than the degree of the representation. However the k-th symmetric power is nonzero for all k.

The tensor square of any representation decomposes as the direct sum of the symmetric and exterior squares:

```
sage: C4 = WeylCharacterRing("C4",style="coroots")
sage: chi = C4(1,0,0,0); chi.degree()
8
sage: chi.symmetric_square()
C4(2,0,0,0)
sage: chi.exterior_square()
C4(0,0,0,0) + C4(0,1,0,0)
sage: chi^2 == chi.symmetric_square() + chi.exterior_square()
True
```

Since in this example the exterior square contains the trivial representation we expect the Frobenius-Schur indicator to be -1, and indeed it is:

```
sage: chi = C4(1,0,0,0)
sage: chi.frobenius_schur_indicator()
-1
```

This is not surprising since this is the standard representation of a symplectic group, which is symplectic *by definition*!

Weyl dimension formula

If the representation is truly large you will not be able to construct it in the Weyl character ring, since internally it is represented by a dictionary of its weights. If you want to know its degree, you can still compute that since Sage implements the Weyl dimension formula. The degree of the representation is implemented as a method of its highest weight vector:

```
sage: L = RootSystem("E8").ambient_space()
sage: [L.weyl_dimension(f) for f in L.fundamental_weights()]
[3875, 147250, 6696000, 6899079264, 146325270, 2450240, 30380, 248]
```

It is a fact that for any compact Lie group if ρ is the Weyl vector (half the sum of the positive roots) then the degree of the irreducible representation with highest weight ρ equals 2^N where N is the number of positive roots. Let us check this for E_8. In this case $N = 120$:

```
sage: L = RootSystem("E8").ambient_space()
sage: len(L.positive_roots())
120
sage: 2^120
1329227995784915872903807060280344576
sage: L.weyl_dimension(L.rho())
1329227995784915872903807060280344576
```

SL versus GL

Sage takes the weight space for type `['A',r]` to be $r+1$ dimensional. As a biproduct, if you create the Weyl character ring with the command:

```
sage: A2 = WeylCharacterRing("A2")
```

Then you are effectively working with $GL(3)$ instead of $SL(3)$. For example, the determinant is the character `A2(1,1,1)`. However, as we will explain later, you can work with $SL(3)$ if you like, so long as you are willing to work with fractional weights. On the other hand if you create the Weyl character ring with the command:

```
sage: A2 = WeylCharacterRing("A2", style="coroots")
```

Then you are working with $SL(3)$.

There are some advantages to this arrangement:

- The group $GL(r+1)$ arises frequently in practice. For example, even if you care mainly about semisimple groups, the group $GL(r+1)$ may arise as a Levi subgroup.
- It avoids fractional weights. If you want to work with $SL(3)$ the fundamental weights are `(2/3,-1/3,-1/3)` and `(1/3,1/3,-2/3)`. If you work instead with $GL(3)$, they are `(1,0,0)` and `(1,1,0)`. For many mathematical purposes it doesn't make any difference which you use. This is because the difference between `(2/3,-1/3,-1/3)` and `(1,0,0)` is a vector that is orthogonal to all the simple roots. Thus these vectors are interchangeable. But for convenience avoiding fractional weights is advantageous.

However if you want to be an SL purist, Sage will support you. The weight space for $SL(3)$ can be taken to be the hyperplane in $\mathbf{Q}^3$ consisting of vectors (a,b,c) with $a+b+c=0$. The fundamental weights for SL(3) are then `(2/3,-1/3,-1/3)` and `(1/3,1/3,-2/3)`, though Sage will tell you they are `(1,0,0)` and `(1,1,0)`. The work-around is to filter them through the method `coerce_to_sl` as follows:

```
sage: A2 = WeylCharacterRing("A2")
sage: [fw1,fw2] = [w.coerce_to_sl() for w in A2.fundamental_weights()]
```

```
sage: [standard, contragredient] = [A2(fw1), A2(fw2)]
sage: standard, contragredient
(A2(2/3,-1/3,-1/3), A2(1/3,1/3,-2/3))
sage: standard*contragredient
A2(0,0,0) + A2(1,0,-1)
```

Sage is not confused by the fractional weights. Note that if you use coroot notation, you are working with SL automatically:

```
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: A2(1,0).weight_multiplicities()
{(-1/3, -1/3, 2/3): 1, (2/3, -1/3, -1/3): 1, (-1/3, 2/3, -1/3): 1}
```

There is no convenient way to create the determinant in the Weyl character ring if you adopt the coroot style.

Just as we coerced the fundamental weights into the SL weight lattice, you may need to coerce the Weyl vector ρ if you are working with SL. The default value for ρ in type A_r is $(r, r-1, \ldots, 0)$, but if you are an SL purist you want

$$\left(\frac{r}{2}, \frac{r}{2} - 1, \ldots, -\frac{r}{2}\right).$$

Therefore take the value of ρ that you get from the method of the ambient space and coerce it into SL:

```
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: rho = A2.space().rho().coerce_to_sl(); rho
(1, 0, -1)
sage: rho == (1/2)*sum(A2.space().positive_roots())
True
```

You do not need to do this for other Cartan types. If you are working with (say) $F4$ then a ρ is a ρ:

```
sage: F4 = WeylCharacterRing("F4")
sage: L = F4.space()
sage: rho = L.rho()
sage: rho == (1/2)*sum(L.positive_roots())
True
```

Integration

Suppose that we wish to compute the integral

$$\int_{U(n)} |tr(g)|^{2k}\, dg$$

for various n. Here $U(n)$ is the unitary group, which is the maximal compact subroup of $GL(n, \mathbf{C})$, and dg is the Haar measure on $U(n)$, normalized so that the volume of the group is 1.

The irreducible unitary representations of $U(n)$ may be regarded as the basis elements of the WeylCharacterRing of type A_r, where $r = n - 1$ so we might work in that ring. The trace $tr(g)$ is then just the character of the standard representation. We may realize it in the WeylCharacterRing by taking the first fundamental weight and coercing it into the ring. For example, if $k = 5$ and $n = 3$ so $r = 2$:

```
sage: A2 = WeylCharacterRing("A2")
sage: fw = A2.fundamental_weights(); fw
Finite family {1: (1, 0, 0), 2: (1, 1, 0)}
sage: tr = A2(fw[1]); tr
A2(1,0,0)
```

We may compute the norm square the character `tr^5` by decomposing it into irreducibles, and taking the sum of the squares of their multiplicities. By Schur orthogonality, this gives the inner product of the $tr(g)^5$ with itself, that is, the integral of $|tr(g)|^{10}$:

```
sage: tr^5
5*A2(2,2,1) + 6*A2(3,1,1) + 5*A2(3,2,0) + 4*A2(4,1,0) + A2(5,0,0)
sage: (tr^5).monomials()
[A2(2,2,1), A2(3,1,1), A2(3,2,0), A2(4,1,0), A2(5,0,0)]
sage: (tr^5).coefficients()
[5, 6, 5, 4, 1]
sage: sum(x^2 for x in (tr^5).coefficients())
103
```

So far we have been working with $n = 3$. For general n:

```
sage: def f(n,k):
....:     R = WeylCharacterRing(['A',n-1])
....:     tr = R(R.fundamental_weights()[1])
....:     return sum(x^2 for x in (tr^k).coefficients())
....:
sage: [f(n,5) for n in [2..7]]  # long time (31s on sage.math, 2012)
[42, 103, 119, 120, 120, 120]
```

We see that the 10-th moment of $tr(g)$ is just 5! when n is sufficiently large. What if we fix n and vary k?

```
sage: [f(2,k) for k in [1..10]]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796]
sage: [catalan_number(k) for k in [1..10]]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796]
```

Invariants and multiplicities

Sometimes we are only interested in the multiplicity of the trivial representation in some character. This may be found by the method `invariant_degree`. Continuing from the preceding example,

```
sage: A2 = WeylCharacterRing("A2",style="coroots")
sage: ad = A2(1,1)
sage: [ad.symmetric_power(k).invariant_degree() for k in [0..6]]
[1, 0, 1, 1, 1, 1, 2]
sage: [ad.exterior_power(k).invariant_degree() for k in [0..6]]
[1, 0, 0, 1, 0, 1, 0]
```

If we want the multiplicity of some other representation, we may obtain that using the method `multiplicity`:

```
sage: (ad^3).multiplicity(ad)
8
```

Maximal Subgroups and Branching Rules

Branching rules

If G is a Lie group and H is a subgroup, one often needs to know how representations of G restrict to H. Irreducibles usually do not restrict to irreducibles. In some cases the restriction is regular and predictable, in other cases it is chaotic. In some cases it follows a rule that can be described combinatorially, but the combinatorial description is subtle. The description of how irreducibles decompose into irreducibles is called a *branching rule*.

References for this topic:

- [FauserEtAl2006]
- [King1975]
- [HoweEtAl2005]
- [McKayPatera1981]

Sage can compute how a character of G restricts to H. It does so not by memorizing a combinatorial rule, but by computing the character and restricting the character to a maximal torus of H. What Sage has memorized (in a series of built-in encoded rules) are the various embeddings of maximal tori of maximal subgroups of G. The maximal subgroups of Lie groups were determined in [Dynkin1952]. This approach to computing branching rules has a limitation: the character must fit into memory and be computable by Sage's internal code in real time.

It is sufficient to consider the case where H is a maximal subgroup of G, since if this is known then one may branch down successively through a series of subgroups, each maximal in its predecessors. A problem is therefore to understand the maximal subgroups in a Lie group, and to give branching rules for each, and a goal of this tutorial is to explain the embeddings of maximal subgroups.

Sage has a class `BranchingRule` for branching rules. The function `branching_rule` returns elements of this class. For example, the natural embedding of $Sp(4)$ into $SL(4)$ corresponds to the branching rule that we may create as follows:

```
sage: b=branching_rule("A3","C2",rule="symmetric"); b
symmetric branching rule A3 => C2
```

The name "symmetric" of this branching rule will be explained further later, but it means that $Sp(4)$ is the fixed subgroup of an involution of $Sl(4)$. Here `A3` and `C2` are the Cartan types of the groups $G = SL(4)$ and $H = Sp(4)$.

Now we may see how representations of $SL(4)$ decompose into irreducibles when they are restricted to $Sp(4)$:

```
sage: A3=WeylCharacterRing("A3",style="coroots")
sage: chi=A3(1,0,1); chi.degree()
15
sage: C2=WeylCharacterRing("C2",style="coroots")
sage: chi.branch(C2,rule=b)
C2(0,1) + C2(2,0)
```

Alternatively, we may pass `chi` to `b` as an argument of its branch method, which gives the same result:

```
sage: b.branch(chi)
C2(0,1) + C2(2,0)
```

It is believed that the built-in branching rules of Sage are sufficient to handle all maximal subgroups and this is certainly the case when the rank if less than or equal to 8.

However, if you want to branch to a subgroup that is not maximal you may not find a built-in branching rule. We may compose branching rules to build up embeddings. For example, here are two different embeddings of $Sp(4)$ with Cartan type `C2` in $Sp(8)$, with Cartan type `C4`. One embedding factors through $Sp(4) \times Sp(4)$, while the other factors through $SL(4)$. To check that the embeddings are not conjugate, we branch a (randomly chosen) representation. Observe that we do not have to build the intermediate `Weyl character rings`.

```
sage: C4=WeylCharacterRing("C4",style="coroots")
sage: b1=branching_rule("C4","A3","levi")*branching_rule("A3","C2","symmetric"); b1
composite branching rule C4 => (levi) A3 => (symmetric) C2
sage: b2=branching_rule("C4","C2xC2","orthogonal_sum")*branching_rule("C2xC2","C2","proj1"); b2
composite branching rule C4 => (orthogonal_sum) C2xC2 => (proj1) C2
sage: C2=WeylCharacterRing("C2",style="coroots")
sage: C4=WeylCharacterRing("C4",style="coroots")
```

```
sage: [C4(2,0,0,1).branch(C2, rule=br) for br in [b1,b2]]
[4*C2(0,0) + 7*C2(0,1) + 15*C2(2,0) + 7*C2(0,2) + 11*C2(2,1) + C2(0,3) + 6*C2(4,0) + 3*C2(2,2),
 10*C2(0,0) + 40*C2(1,0) + 50*C2(0,1) + 16*C2(2,0) + 20*C2(1,1) + 4*C2(3,0) + 5*C2(2,1)]
```

What's in a branching rule?

The essence of the branching rule is a function from the weight lattice of G to the weight lattice of the subgroup H, usually implemented as a function on the ambient vector spaces. Indeed, we may conjugate the embedding so that a Cartan subalgebra U of H is contained in a Cartan subalgebra T of G. Since the ambient vector space of the weight lattice of G is $\mathrm{Lie}(T)^*$, we get map $\mathrm{Lie}(T)^* \to \mathrm{Lie}(U)^*$, and this must be implemented as a function. For speed, the function usually just returns a list, which can be coerced into $\mathrm{Lie}(U)^*$.

```
sage: b = branching_rule("A3","C2","symmetric")
sage: for r in RootSystem("A3").ambient_space().simple_roots(): print r, b(r)
(1, -1, 0, 0) [1, -1]
(0, 1, -1, 0) [0, 2]
(0, 0, 1, -1) [1, -1]
```

We could conjugate this map by an element of the Weyl group of G, and the resulting map would give the same decomposition of representations of G into irreducibles of H. However it is a good idea to choose the map so that it takes dominant weights to dominant weights, and, insofar as possible, simple roots of G into simple roots of H. This includes sometimes the affine root α_0 of G, which we recall is the negative of the highest root.

The branching rule has a `describe()` method that shows how the roots (including the affine root) restrict. This is a useful way of understanding the embedding. You might want to try it with various branching rules of different kinds, `"extended"`, `"symmetric"`, `"levi"` etc.

```
sage: b.describe()

0
O-------+
|       |
|       |
O---O---O
1   2   3
A3~

root restrictions A3 => C2:

O=<=O
1   2
C2

1 => 1
2 => 2
3 => 1

For more detailed information use verbose=True
```

The extended Dynkin diagram of G and the ordinary Dynkin diagram of H are shown for reference, and `3 => 1` means that the third simple root α_3 of G restricts to the first simple root of H. In this example, the affine root does not restrict to a simple roots, so it is omitted from the list of restrictions. If you add the parameter `verbose=true` you will be shown the restriction of all simple roots and the affine root, and also the restrictions of the fundamental weights (in coroot notation).

Maximal subgroups

Sage has a database of maximal subgroups for every simple Cartan type of rank ≤ 8. You may access this with the `maximal_subgroups` method of the Weyl character ring:

```
sage: E7=WeylCharacterRing("E7",style="coroots")
sage: E7.maximal_subgroups()
A7:branching_rule("E7","A7","extended")
E6:branching_rule("E7","E6","levi")
A2:branching_rule("E7","A2","miscellaneous")
A1:branching_rule("E7","A1","iii")
A1:branching_rule("E7","A1","iv")
A1xF4:branching_rule("E7","A1xF4","miscellaneous")
G2xC3:branching_rule("E7","G2xC3","miscellaneous")
A1xG2:branching_rule("E7","A1xG2","miscellaneous")
A1xA1:branching_rule("E7","A1xA1","miscellaneous")
A1xD6:branching_rule("E7","A1xD6","extended")
A5xA2:branching_rule("E7","A5xA2","extended")
```

It should be understood that there are other ways of embedding $A_2 = \mathrm{SL}(3)$ into the Lie group E_7, but only one way as a maximal subgroup. On the other hand, there are but only one way to embed it as a maximal subgroup. The embedding will be explained below. You may obtain the branching rule as follows, and use it to determine the decomposition of irreducible representations of E_7 as follows:

```
sage: b = E7.maximal_subgroup("A2"); b
miscellaneous branching rule E7 => A2
sage: [E7,A2]=[WeylCharacterRing(x,style="coroots") for x in ["E7","A2"]]
sage: E7(1,0,0,0,0,0,0).branch(A2,rule=b)
A2(1,1) + A2(4,4)
```

This gives the same branching rule as just pasting line beginning to the right of the colon onto the command line:

```
sage:branching_rule("E7","A2","miscellaneous")
miscellaneous branching rule E7 => A2
```

There are two distict embeddings of $A_1 = \mathrm{SL}(2)$ into E_7 as maximal subgroups, so the `maximal_subgroup` method will return a list of rules:

```
sage: WeylCharacterRing("E7").maximal_subgroup("A1")
[iii branching rule E7 => A1, iv branching rule E7 => A1]
```

The list of maximal subgroups returned by the `maximal_subgroups` method for irreducible Cartan types of rank up to 8 is believed to be complete up to outer automorphisms. You may want a list that is complete up to inner automorphisms. For example, E_6 has a nontrivial Dynkin diagram automorphism so it has an outer automorphism that is not inner:

```
sage: [E6,A2xG2]=[WeylCharacterRing(x,style="coroots") for x in ["E6","A2xG2"]]
sage: b=E6.maximal_subgroup("A2xG2"); b
miscellaneous branching rule E6 => A2xG2
sage: E6(1,0,0,0,0,0).branch(A2xG2,rule=b)
A2xG2(0,1,1,0) + A2xG2(2,0,0,0)
sage: E6(0,0,0,0,0,1).branch(A2xG2,rule=b)
A2xG2(1,0,1,0) + A2xG2(0,2,0,0)
```

Since as we see the two 27 dimensional irreducibles (which are interchanged by the outer automorphism) have different branching, the $A_2 \times G_2$ subgroup is changed to a different one by the outer automorphism. To obtain the second branching rule, we compose the given one with this automorphism:

```
sage: b1=branching_rule("E6","E6","automorphic")*b; b1
composite branching rule E6 => (automorphic) E6 => (miscellaneous) A2xG2
```

Levi subgroups

A Levi subgroup may or may not be maximal. They are easily classified. If one starts with a Dynkin diagram for G and removes a single node, one obtains a smaller Dynkin diagram, which is the Dynkin diagram of a smaller subgroup H.

For example, here is the A3 Dynkin diagram:

```
sage: A3 = WeylCharacterRing("A3")
sage: A3.dynkin_diagram()
O---O---O
1   2   3
A3
```

We see that we may remove the node 3 and obtain A_2, or the node 2 and obtain $A_1 \times A_1$. These correspond to the Levi subgroups $GL(3)$ and $GL(2) \times GL(2)$ of $GL(4)$.

Let us construct the irreducible representations of $GL(4)$ and branch them down to these down to $GL(3)$ and $GL(2) \times GL(2)$:

```
sage: reps = [A3(v) for v in A3.fundamental_weights()]; reps
[A3(1,0,0,0), A3(1,1,0,0), A3(1,1,1,0)]
sage: A2 = WeylCharacterRing("A2")
sage: A1xA1 = WeylCharacterRing("A1xA1")
sage: [pi.branch(A2, rule="levi") for pi in reps]
[A2(0,0,0) + A2(1,0,0), A2(1,0,0) + A2(1,1,0), A2(1,1,0) + A2(1,1,1)]
sage: [pi.branch(A1xA1, rule="levi") for pi in reps]
[A1xA1(1,0,0,0) + A1xA1(0,0,1,0),
 A1xA1(1,1,0,0) + A1xA1(1,0,1,0) + A1xA1(0,0,1,1),
 A1xA1(1,1,1,0) + A1xA1(1,0,1,1)]
```

Let us redo this calculation in coroot notation. As we have explained, coroot notation does not distinguish between representations of $GL(4)$ that have the same restriction to $SL(4)$, so in effect we are now working with the groups $SL(4)$ and its Levi subgroups $SL(3)$ and $SL(2) \times SL(2)$, which is the derived group of its Levi subgroup:

```
sage: A3 = WeylCharacterRing("A3", style="coroots")
sage: reps = [A3(v) for v in A3.fundamental_weights()]; reps
[A3(1,0,0), A3(0,1,0), A3(0,0,1)]
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: A1xA1 = WeylCharacterRing("A1xA1", style="coroots")
sage: [pi.branch(A2, rule="levi") for pi in reps]
[A2(0,0) + A2(1,0), A2(0,1) + A2(1,0), A2(0,0) + A2(0,1)]
sage: [pi.branch(A1xA1, rule="levi") for pi in reps]
[A1xA1(1,0) + A1xA1(0,1), 2*A1xA1(0,0) + A1xA1(1,1), A1xA1(1,0) + A1xA1(0,1)]
```

Now we may observe a distinction difference in branching from

$$GL(4) \to GL(2) \times GL(2)$$

versus

$$SL(4) \to SL(2) \times SL(2).$$

Consider the representation `A3(0,1,0)`, which is the six dimensional exterior square. In the coroot notation, the restriction contained two copies of the trivial representation, `2*A1xA1(0,0)`. The other way, we had instead three

distinct representations in the restriction, namely `A1xA1(1,1,0,0)` and `A1xA1(0,0,1,1)`, that is, $\det \otimes 1$ and $1 \otimes \det$.

The Levi subgroup `A1xA1` is actually not maximal. Indeed, we may factor the embedding:

$$SL(2) \times SL(2) \to Sp(4) \to SL(4).$$

Therfore there are branching rules `A3 -> C2` and `C2 -> A2`, and we could accomplish the branching in two steps, thus:

```
sage: A3 = WeylCharacterRing("A3", style="coroots")
sage: C2 = WeylCharacterRing("C2", style="coroots")
sage: B2 = WeylCharacterRing("B2", style="coroots")
sage: D2 = WeylCharacterRing("D2", style="coroots")
sage: A1xA1 = WeylCharacterRing("A1xA1", style="coroots")
sage: reps = [A3(fw) for fw in A3.fundamental_weights()]
sage: [pi.branch(C2, rule="symmetric").branch(B2, rule="isomorphic"). \
         branch(D2, rule="extended").branch(A1xA1, rule="isomorphic") for pi in reps]
[A1xA1(1,0) + A1xA1(0,1), 2*A1xA1(0,0) + A1xA1(1,1), A1xA1(1,0) + A1xA1(0,1)]
```

As you can see, we've redone the branching rather circuitously this way, making use of the branching rules `A3 -> C2` and `B2 -> D2`, and two accidental isomorphisms `C2 = B2` and `D2 = A1xA1`. It is much easier to go in one step using `rule="levi"`, but reassuring that we get the same answer!

Subgroups classified by the extended Dynkin diagram

It is also true that if we remove one node from the extended Dynkin diagram that we obtain the Dynkin diagram of a subgroup. For example:

```
sage: G2 = WeylCharacterRing("G2", style="coroots")
sage: G2.extended_dynkin_diagram()
  3
O=<=O---O
1   2   0
G2~
```

Observe that by removing the 1 node that we obtain an A_2 Dynkin diagram. Therefore the exceptional group G_2 contains a copy of $SL(3)$. We branch the two representations of G_2 corresponding to the fundamental weights to this copy of A_2:

```
sage: G2 = WeylCharacterRing("G2", style="coroots")
sage: A2 = WeylCharacterRing("A2", style="coroots")
sage: [G2(f).degree() for f in G2.fundamental_weights()]
[7, 14]
sage: [G2(f).branch(A2, rule="extended") for f in G2.fundamental_weights()]
[A2(0,0) + A2(0,1) + A2(1,0), A2(0,1) + A2(1,0) + A2(1,1)]
```

The two representations of G_2, of degrees 7 and 14 respectively, are the action on the octonions of trace zero and the adjoint representation.

For embeddings of this type, the rank of the subgroup H is the same as the rank of G. This is in contrast with embeddings of Levi type, where H has rank one less than G.

Levi subgroups of G_2

The exceptional group G_2 has two Levi subgroups of type A_1. Neither is maximal, as we can see from the extended Dynkin diagram: the subgroups $A_1 \times A_1$ and A_2 are maximal and each contains a Levi subgroup. (Actually $A_1 \times A_1$

contains a conjugate of both.) Only the Levi subgroup containing the short root is implemented as an instance of `rule="levi"`. To obtain the other, use the rule:

```
sage: branching_rule("G2","A2","extended")*branching_rule("A2","A1","levi")
composite branching rule G2 => (extended) A2 => (levi) A1
```

which branches to the A_1 Levi subgroup containing a long root.

Orthogonal and symplectic subgroups of orthogonal and symplectic groups

If $G = \mathrm{SO}(n)$ then G has a subgroup $\mathrm{SO}(n-1)$. Depending on whether n is even or odd, we thus have branching rules `['D',r]` to `['B',r-1]` or `['B',r]` to `['D',r]`. These are handled as follows:

```
sage: branching_rule("B4","D4",rule="extended")
extended branching rule B4 => D4
sage: branching_rule("D4","B3",rule="symmetric")
symmetric branching rule D4 => B3
```

If $G = \mathrm{SO}(r+s)$ then G has a subgroup $\mathrm{SO}(r) \times \mathrm{SO}(s)$. This lifts to an embedding of the universal covering groups

$$\mathrm{spin}(r) \times \mathrm{spin}(s) \to \mathrm{spin}(r+s).$$

Sometimes this embedding is of extended type, and sometimes it is not. It is of extended type unless r and s are both odd. If it is of extended type then you may use `rule="extended"`. In any case you may use `rule="orthogonal_sum"`. The name refer to the origin of the embedding $SO(r) \times SO(s) \to SO(r+s)$ from the decomposition of the underlying quadratic space as a direct sum of two orthogonal subspaces.

There are four cases depending on the parity of r and s. For example, if $r = 2k$ and $s = 2l$ we have an embedding:

```
['D',k] x ['D',l] --> ['D',k+l]
```

This is of extended type. Thus consider the embedding `D4xD3 -> D7`. Here is the extended Dynkin diagram:

```
  0 O             O 7
    |             |
    |             |
O---O---O---O---O---O
1   2   3   4   5   6
```

Removing the 4 vertex results in a disconnected Dynkin diagram:

```
  0 O             O 7
    |             |
    |             |
O---O---O         O---O
1   2   3         5   6
```

This is `D4xD3`. Therefore use the "extended" branching rule:

```
sage: D7 = WeylCharacterRing("D7", style="coroots")
sage: D4xD3 = WeylCharacterRing("D4xD3", style="coroots")
sage: spin = D7(D7.fundamental_weights()[7]); spin
D7(0,0,0,0,0,0,1)
sage: spin.branch(D4xD3, rule="extended")
D4xD3(0,0,1,0,0,1,0) + D4xD3(0,0,0,1,0,0,1)
```

But we could equally well use the "orthogonal_sum" rule:

```
sage: spin.branch(D4xD3, rule="orthogonal_sum")
D4xD3(0,0,1,0,0,1,0) + D4xD3(0,0,0,1,0,0,1)
```

Similarly we have embeddings:

```
['D',k] x ['B',l] --> ['B',k+l]
```

These are also of extended type. For example consider the embedding of `D3xB2 -> B5`. Here is the `B5` extended Dynkin diagram:

```
    O 0
    |
    |
O---O---O---O=>=O
1   2   3   4   5
```

Removing the 3 node gives:

```
    O 0
    |
O---O       O=>=O
1   2       4   5
```

and this is the Dynkin diagram or `D3xB2`. For such branchings we again use either `rule="extended"` or `rule="orthogonal_sum"`.

Finally, there is an embedding

```
['B',k] x ['B',l] --> ['D',k+l+1]
```

This is *not* of extended type, so you may not use `rule="extended"`. You *must* use `rule="orthogonal_sum"`:

```
sage: D5 = WeylCharacterRing("D5",style="coroots")
sage: B2xB2 = WeylCharacterRing("B2xB2",style="coroots")
sage: [D5(v).branch(B2xB2,rule="orthogonal_sum") for v in D5.fundamental_weights()]
[B2xB2(1,0,0,0) + B2xB2(0,0,1,0),
 B2xB2(0,2,0,0) + B2xB2(1,0,1,0) + B2xB2(0,0,0,2),
 B2xB2(0,2,0,0) + B2xB2(0,2,1,0) + B2xB2(1,0,0,2) + B2xB2(0,0,0,2),
 B2xB2(0,1,0,1), B2xB2(0,1,0,1)]
```

Non-maximal Levi subgroups and Projection from Reducible Types

Not all Levi subgroups are maximal. Recall that the Dynkin-diagram of a Levi subgroup H of G is obtained by removing a node from the Dynkin diagram of G. Removing the same node from the extended Dynkin diagram of G results in the Dynkin diagram of a subgroup of G that is strictly larger than H. However this subgroup may or may not be proper, so the Levi subgroup may or may not be maximal.

If the Levi subgroup is not maximal, the branching rule may or may not be implemented in Sage. However if it is not implemented, it may be constructed as a composition of two branching rules.

For example, prior to Sage-6.1 `branching_rule("E6","A5","levi")` returned a not-implemented error and the advice to branch to `A5xA1`. And we can see from the extended Dynkin diagram of E_6 that indeed A_5 is not a maximal subgroup, since removing node 2 from the extended Dynkin diagram (see below) gives `A5xA1`. To construct the branching rule to A_5 we may proceed as follows:

```
sage: b = branching_rule("E6","A5xA1","extended")*branching_rule("A5xA1","A5","proj1"); b
composite branching rule E6 => (extended) A5xA1 => (proj1) A5
sage: E6=WeylCharacterRing("E6",style="coroots")
sage: A5=WeylCharacterRing("A5",style="coroots")
sage: E6(0,1,0,0,0,0).branch(A5,rule=b)
3*A5(0,0,0,0,0) + 2*A5(0,0,1,0,0) + A5(1,0,0,0,1)
sage: b.describe()

        O 0
        |
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
root restrictions E6 => A5:

O---O---O---O---O
1   2   3   4   5
A5

0 => (zero)
1 => 1
3 => 2
4 => 3
5 => 4
6 => 5

For more detailed information use verbose=True
```

Note that it is not necessary to construct the Weyl character ring for the intermediate group `A5xA1`.

This last example illustrates another common problem: how to extract one component from a reducible root system. We used the rule `"proj1"` to extract the first component. We could similarly use `"proj2"` to get the second, or more generally any combination of components:

```
sage: branching_rule("A2xB2xG2","A2xG2","proj13")
proj13 branching rule A2xB2xG2 => A2xG2
```

Symmetric subgroups

If G admits an outer automorphism (usually of order two) then we may try to find the branching rule to the fixed subgroup H. It can be arranged that this automorphism maps the maximal torus T to itself and that a maximal torus U of H is contained in T.

Suppose that the Dynkin diagram of G admits an automorphism. Then G itself admits an outer automorphism. The Dynkin diagram of the group H of invariants may be obtained by "folding" the Dynkin diagram of G along the automorphism. The exception is the branching rule $GL(2r) \to SO(2r)$.

Here are the branching rules that can be obtained using `rule="symmetric"`.

G	H	Cartan Types
$GL(2r)$	$Sp(2r)$	`['A',2r-1] => ['C',r]`
$GL(2r+1)$	$SO(2r+1)$	`['A',2r] => ['B',r]`
$GL(2r)$	$SO(2r)$	`['A',2r-1] => ['D',r]`
$SO(2r)$	$SO(2r-1)$	`['D',r] => ['B',r-1]`
E_6	F_4	`['E',6] => ['F',4]`

Tensor products

If G_1 and G_2 are Lie groups, and we have representations $\pi_1 : G_1 \to GL(n)$ and $\pi_2 : G_2 \to GL(m)$ then the tensor product is a representation of $G_1 \times G_2$. It has its image in $GL(nm)$ but sometimes this is conjugate to a subgroup of $SO(nm)$ or $Sp(nm)$. In particular we have the following cases.

Group	Subgroup	Cartan Types
$GL(rs)$	$GL(r) \times GL(s)$	`['A', rs-1] => ['A',r-1] x ['A',s-1]`
$SO(4rs+2r+2s+1)$	$SO(2r+1) \times SO(2s+1)$	`['B',2rs+r+s] => ['B',r] x ['B',s]`
$SO(4rs+2s)$	$SO(2r+1) \times SO(2s)$	`['D',2rs+s] => ['B',r] x ['D',s]`
$SO(4rs)$	$SO(2r) \times SO(2s)$	`['D',2rs] => ['D',r] x ['D',s]`
$SO(4rs)$	$Sp(2r) \times Sp(2s)$	`['D',2rs] => ['C',r] x ['C',s]`
$Sp(4rs+2s)$	$SO(2r+1) \times Sp(2s)$	`['C',2rs+s] => ['B',r] x ['C',s]`
$Sp(4rs)$	$Sp(2r) \times SO(2s)$	`['C',2rs] => ['C',r] x ['D',s]`

These branching rules are obtained using `rule="tensor"`.

Symmetric powers

The k-th symmetric and exterior power homomorphisms map $GL(n) \to GL\left(\binom{n+k-1}{k}\right)$ and $GL\left(\binom{n}{k}\right)$. The corresponding branching rules are not implemented but a special case is. The k-th symmetric power homomorphism $SL(2) \to GL(k+1)$ has its image inside of $SO(2r+1)$ if $k = 2r$ and inside of $Sp(2r)$ if $k = 2r-1$. Hence there are branching rules:

```
['B',r] => A1
['C',r] => A1
```

and these may be obtained using `rule="symmetric_power"`.

Plethysms

The above branching rules are sufficient for most cases, but a few fall between the cracks. Mostly these involve maximal subgroups of fairly small rank.

The rule `rule="plethysm"` is a powerful rule that includes any branching rule from types A, B, C or D as a special case. Thus it could be used in place of the above rules and would give the same results. However, it is most useful when branching from G to a maximal subgroup H such that $rank(H) < rank(G) - 1$.

We consider a homomorphism $H \to G$ where G is one of $SL(r+1)$, $SO(2r+1)$, $Sp(2r)$ or $SO(2r)$. The function `branching_rule_from_plethysm` produces the corresponding branching rule. The main ingredient is the character χ of the representation of H that is the homomorphism to $GL(r+1)$, $GL(2r+1)$ or $GL(2r)$.

Let us consider the symmetric fifth power representation of $SL(2)$. This is implemented above by `rule="symmetric_power"`, but suppose we want to use `rule="plethysm"`. First we construct the homomorphism by invoking its character, to be called `chi`:

```
sage: A1 = WeylCharacterRing("A1", style="coroots")
sage: chi = A1([5])
sage: chi.degree()
6
sage: chi.frobenius_schur_indicator()
-1
```

This confirms that the character has degree 6 and is symplectic, so it corresponds to a homomorphism $SL(2) \to Sp(6)$, and there is a corresponding branching rule `C3 -> A1`:

```
sage: A1 = WeylCharacterRing("A1", style="coroots")
sage: C3 = WeylCharacterRing("C3", style="coroots")
sage: chi = A1([5])
sage: sym5rule = branching_rule_from_plethysm(chi, "C3")
sage: [C3(hwv).branch(A1, rule=sym5rule) for hwv in C3.fundamental_weights()]
[A1(5), A1(4) + A1(8), A1(3) + A1(9)]
```

This is identical to the results we would obtain using `rule="symmetric_power"`:

```
sage: A1 = WeylCharacterRing("A1", style="coroots")
sage: C3 = WeylCharacterRing("C3", style="coroots")
sage: [C3(v).branch(A1, rule="symmetric_power") for v in C3.fundamental_weights()]
[A1(5), A1(4) + A1(8), A1(3) + A1(9)]
```

But the next example of plethysm gives a branching rule not available by other methods:

```
sage: G2 = WeylCharacterRing("G2", style="coroots")
sage: D7 = WeylCharacterRing("D7", style="coroots")
sage: ad = G2.adjoint_representation(); ad.degree()
14
sage: ad.frobenius_schur_indicator()
1
sage: for r in D7.fundamental_weights():  # long time (1.29s)
....:     print D7(r).branch(G2, rule=branching_rule_from_plethysm(ad, "D7"))
....:
G2(0,1)
G2(0,1) + G2(3,0)
G2(0,0) + G2(2,0) + G2(3,0) + G2(0,2) + G2(4,0)
G2(0,1) + G2(2,0) + G2(1,1) + G2(0,2) + G2(2,1) + G2(4,0) + G2(3,1)
G2(1,0) + G2(0,1) + G2(1,1) + 2*G2(3,0) + 2*G2(2,1) + G2(1,2) + G2(3,1) + G2(5,0) + G2(0,3)
G2(1,1)
G2(1,1)
```

In this example, ad is the 14-dimensional adjoint representation of the exceptional group G_2. Since the Frobenius-Schur indicator is 1, the representation is orthogonal, and factors through $SO(14)$, that is, $D7$.

We do not actually have to create the character (or for that matter its ambient WeylCharacterRing) in order to create the branching rule:

```
sage: branching_rule("D4","A2.adjoint_representation()","plethysm")
plethysm (along A2(1,1)) branching rule D4 => A2
```

The adjoint representation of any semisimple Lie group is orthogonal, so we do not need to compute the Frobenius-Schur indicator.

Miscellaneous other subgroups

Use `rule="miscellaneous"` for the following rules. Every maximal subgroup H of an exceptional group G are either among these, or the five A_1 subgroups described in the next section, or (if G and H have the same rank) is available using `rule="extended"`.

$$
\begin{aligned}
&B_3 \to G_2,\\
&E_6 \to A_2,\\
&E_6 \to G_2,\\
&F_4 \to G_2 \times A_1,\\
&E_6 \to G_2 \times A_2,\\
&E_7 \to G_2 \times C_3,\\
&E_7 \to F_4 \times A_1,\\
&E_7 \to A_1 \times A_1,\\
&E_7 \to G_2 \times A_1,\\
&E_7 \to A_2\\
&E_8 \to G_2 \times F_4.\\
&E_8 \to A_2 \times A_1.\\
&E_8 \to B_2
\end{aligned}
$$

The first rule corresponds to the embedding of G_2 in $\mathrm{SO}(7)$ in its action on the trace zero octonions. The two branching rules from E_6 to G_2 or A_2 are described in [Testerman1989]. We caution the reader that Theorem G.2 of that paper, proved there in positive characteristic is false over the complex numbers. On the other hand, the assumption of characteristic p is not important for Theorems G.1 and A.1, which describe the torus embeddings, hence contain enough information to compute the branching rule. There are other ways of embedding G_2 or A_2 into E_6. These may embeddings be characterized by the condition that the two 27-dimensional representations of E_6 restrict irreducibly to G_2 or A_2. Their images are maximal subgroups.

The remaining rules come about as follows. Let G be F_4, E_6, E_7 or E_8, and let H be G_2, or else (if $G = E_7$) F_4. We embed H into G in the most obvious way; that is, in the chain of subgroups

$$G_2 \subset F_4 \subset E_6 \subset E_7 \subset E_8$$

Then the centralizer of H is A_1, A_2, C_3, F_4 (if $H = G_2$) or A_1 (if $G = E_7$ and $H = F_4$). This gives us five of the cases. Regarding the branching rule $E_6 \to G_2 \times A_2$, Rubenthaler [Rubenthaler2008] describes the embedding and applies it in an interesting way.

The embedding of $A_1 \times A_1$ into E_7 is as follows. Deleting the 5 node of the E_7 Dynkin diagram gives the Dynkin diagram of $A_4 \times A_2$, so this is a Levi subgroup. We embed $\mathrm{SL}(2)$ into this Levi subgroup via the representation $[4] \otimes [2]$. This embeds the first copy of A_1. The other A_1 is the connected centralizer. See [Seitz1991], particularly the proof of (3.12).

The embedding if $G_2 \times A_1$ into E_7 is as follows. Deleting the 2 node of the E_7 Dynkin diagram gives the A_6 Dynkin diagram, which is the Levi subgroup $\mathrm{SL}(7)$. We embed G_2 into $\mathrm{SL}(7)$ via the irreducible seven-dimensional representation of G_2. The A_1 is the centralizer.

The embedding if $A_2 \times A_1$ into E_8 is as follows. Deleting the 2 node of the E_8 Dynkin diagram gives the A_7 Dynkin diagram, which is the Levi subgroup $\mathrm{SL}(8)$. We embed A_2 into $\mathrm{SL}(8)$ via the irreducible eight-dimensional adjoint representation of $\mathrm{SL}(2)$. The A_1 is the centralizer.

The embedding A_2 into E_7 is proved in [Seitz1991] (5.8). In particular, he computes the embedding of the SL(3) torus in the E_7 torus, which is what is needed to implement the branching rule. The embedding of B_2 into E_8 is also constructed in [Seitz1991] (6.7). The embedding of the B_2 Cartan subalgebra, needed to implement the branching rule, is easily deduced from (10) on page 111.

Maximal A1 subgroups of Exceptional Groups

There are seven embeddings of $SL(2)$ into an exceptional group as a maximal subgroup: one each for G_2 and F_4, two nonconjugate embeddings for E_7 and three for E_8 These are constructed in [Testerman1992]. Create the corresponding branching rules as follows. The names of the rules are roman numerals referring to the seven cases of Testerman's Theorem 1:

```
sage: branching_rule("G2","A1","i")
i branching rule G2 => A1
sage: branching_rule("F4","A1","ii")
ii branching rule F4 => A1
sage: branching_rule("E7","A1","iii")
iii branching rule E7 => A1
sage: branching_rule("E7","A1","iv")
iv branching rule E7 => A1
sage: branching_rule("E8","A1","v")
v branching rule E8 => A1
sage: branching_rule("E8","A1","vi")
vi branching rule E8 => A1
sage: branching_rule("E8","A1","vii")
vii branching rule E8 => A1
```

The embeddings are characterized by the root restrictions in their branching rules: usually a simple root of the ambient group G restricts to the unique simple root of A_1, except for root α_4 for rules iv, vi and vii, and the root α_6 for root vii; this is essentially the way Testerman characterizes the embeddings, and this information may be obtained from Sage by employing the `describe()` method of the branching rule. Thus:

```
sage: branching_rule("E8","A1","vii").describe()
```

```
        O 2
        |
        |
O---O---O---O---O---O---O---O
1   3   4   5   6   7   8   0
E8~
root restrictions E8 => A1:

O
1
A1

1 => 1
2 => 1
3 => 1
4 => (zero)
5 => 1
6 => (zero)
7 => 1
8 => 1

For more detailed information use verbose=True
```

Writing your own branching rules

Sage has many built-in branching rules. Indeed, at least up to rank eight (including all the exceptional groups) branching rules to all maximal subgroups are implemented as built in rules, except for a few obtainable using `branching_rule_from_plethysm`. This means that all the rules in [McKayPatera1981] are available in Sage.

Still in this section we are including instructions for coding a rule by hand. As we have already explained, the branching rule is a function from the weight lattice of `G` to the weight lattice of `H`, and if you supply this you can write your own branching rules.

As an example, let us consider how to implement the branching rule `A3 -> C2`. Here `H = C2 = Sp(4)` embedded as a subgroup in `A3 = GL(4)`. The Cartan subalgebra Lie(U) consists of diagonal matrices with eigenvalues `u1, u2, -u2, -u1`. Then `C2.space()` is the two dimensional vector spaces consisting of the linear functionals `u1` and `u2` on `U`. On the other hand $Lie(T) = \mathbf{R}^4$. A convenient way to see the restriction is to think of it as the adjoint of the map `[u1,u2] -> [u1,u2,-u2,-u1]`, that is, `[x0,x1,x2,x3] -> [x0-x3,x1-x2]`. Hence we may encode the rule:

```
def brule(x):
    return [x[0]-x[3], x[1]-x[2]]
```

or simply:

```
brule = lambda x: [x[0]-x[3], x[1]-x[2]]
```

Let us check that this agrees with the built-in rule:

```
sage: A3 = WeylCharacterRing(['A', 3])
sage: C2 = WeylCharacterRing(['C', 2])
sage: brule = lambda x: [x[0]-x[3], x[1]-x[2]]
sage: A3(1,1,0,0).branch(C2, rule=brule)
C2(0,0) + C2(1,1)
sage: A3(1,1,0,0).branch(C2, rule="symmetric")
C2(0,0) + C2(1,1)
```

Although this works, it is better to make the rule into an element of the `BranchingRule` class, as follows.

```
sage: brule = BranchingRule("A3","C2",lambda x : [x[0]-x[3], x[1]-x[2]],"custom")
sage: A3(1,1,0,0).branch(C2, rule=brule)
C2(0,0) + C2(1,1)
```

Automorphisms and triality

The case where $G = H$ can be treated as a special case of a branching rule. In most cases if G has a nontrivial outer automorphism, it has order two, corresponding to the symmetry of the Dynkin diagram. Such an involution exists in the cases A_r, D_r, E_6.

So the automorphism acts on the representations of G, and its effect may be computed using the branching rule code:

```
sage: A4 = WeylCharacterRing("A4",style="coroots")
sage: A4(1,0,1,0).degree()
45
sage: A4(0,1,0,1).degree()
45
sage: A4(1,0,1,0).branch(A4,rule="automorphic")
A4(0,1,0,1)
```

In the special case where G=D4, the Dynkin diagram has extra symmetries, and these correspond to outer automorphisms of the group. These are implemented as the "triality" branching rule:

```
sage: branching_rule("D4","D4","triality").describe()
```

```
    O 4
    |
    |
O---O---O
1   |2  3
    |
    O 0
D4~
root restrictions D4 => D4:

    O 4
    |
    |
O---O---O
1   2   3
D4

1 => 3
2 => 2
3 => 4
4 => 1

For more detailed information use verbose=True
```

Triality his is not an automorphisms of $SO(8)$, but of its double cover $spin(8)$. Note that $spin(8)$ has three representations of degree 8, namely the standard representation of $SO(8)$ and the two eight-dimensional spin representations. These are permuted by triality:

```
sage: D4=WeylCharacterRing("D4",style="coroots")
sage: D4(0,0,0,1).branch(D4,rule="triality")
D4(1,0,0,0)
sage: D4(0,0,0,1).branch(D4,rule="triality").branch(D4,rule="triality")
D4(0,0,1,0)
sage: D4(0,0,0,1).branch(D4,rule="triality").branch(D4,rule="triality").branch(D4,rule="triality")
D4(0,0,0,1)
```

By contrast, rule="automorphic" simply interchanges the two spin representations, as it always does in type D:

```
sage: D4(0,0,0,1).branch(D4,rule="automorphic")
D4(0,0,1,0)
sage: D4(0,0,1,0).branch(D4,rule="automorphic")
D4(0,0,0,1)
```

Weight Rings

You may wish to work directly with the weights of a representation.

Weyl character ring elements are represented internally by a dictionary of their weights with multiplicities. However these are subject to a constraint: the coefficients must be invariant under the action of the Weyl group.

The WeightRing is also a ring whose elements are represented internally by a dictionary of their weights with multiplicities, but it is not subject to this constraint of Weyl group invariance. The weights are allowed to be fractional,

that is, elements of the ambient space. In other words, the weight ring is the group algebra over the ambient space of the weight lattice.

To create a `WeightRing` first construct the `WeylCharacterRing`, then create the `WeightRing` as follows:

```
sage: A2 = WeylCharacterRing(['A',2])
sage: a2 = WeightRing(A2)
```

You may coerce elements of the `WeylCharacterRing` into the weight ring. For example, if you want to see the weights of the adjoint representation of $GL(3)$, you may use the method `mlist`, but another way is to coerce it into the weight ring:

```
sage: from pprint import pprint
sage: A2 = WeylCharacterRing(['A',2])
sage: ad = A2(1,0,-1)
sage: pprint(ad.weight_multiplicities())
{(0, 0, 0): 2, (-1, 1, 0): 1, (-1, 0, 1): 1, (1, -1, 0): 1,
 (1, 0, -1): 1, (0, -1, 1): 1, (0, 1, -1): 1}
```

This command produces a dictionary of the weights that appear in the representation, together with their multiplicities. But another way of getting the same information, with an aim of working with it, is to coerce it into the weight ring:

```
sage: a2 = WeightRing(A2)
sage: a2(ad)
2*a2(0,0,0) + a2(-1,1,0) + a2(-1,0,1) + a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) + a2(0,1,-1)
```

For example, the Weyl denominator formula is usually written this way:

$$\prod_{\alpha\in\Phi^+}\left(e^{\alpha/2}-e^{-\alpha/2}\right)=\sum_{w\in W}(-1)^{l(w)}e^{w(\rho)}.$$

The notation is as follows. Here if λ is a weight, or more generally, an element of the ambient space, then e^λ means the image of λ in the group algebra of the ambient space of the weight lattice λ. Since this group algebra is just the weight ring, we can interpret e^λ as its image in the weight ring.

Let us confirm the Weyl denominator formula for `A2`:

```
sage: A2 = WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: L = A2.space()
sage: W = L.weyl_group()
sage: rho = L.rho().coerce_to_sl()
sage: lhs = prod(a2(alpha/2)-a2(-alpha/2) for alpha in L.positive_roots()); lhs
a2(-1,1,0) - a2(-1,0,1) - a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) - a2(0,1,-1)
sage: rhs = sum((-1)^(w.length())*a2(w.action(rho)) for w in W); rhs
a2(-1,1,0) - a2(-1,0,1) - a2(1,-1,0) + a2(1,0,-1) + a2(0,-1,1) - a2(0,1,-1)
sage: lhs == rhs
True
```

Note that we have to be careful to use the right value of ρ. The reason for this is explained in *SL versus GL*.

We have seen that elements of the `WeylCharacterRing` can be coerced into the `WeightRing`. Elements of the `WeightRing` can be coerced into the `WeylCharacterRing` *provided* they are invariant under the Weyl group.

Weyl Groups, Coxeter Groups and the Bruhat Order

Classical and affine Weyl groups

You can create Weyl groups and affine Weyl groups for any root system. A variety of methods are available for these. Some of these are methods are available for general Coxeter groups.

By default, elements of the Weyl group are represented as matrices:

```
sage: WeylGroup("A3").simple_reflection(1)
[0 1 0 0]
[1 0 0 0]
[0 0 1 0]
[0 0 0 1]
```

You may prefer a notation in which elements are written out as products of simple reflections. In order to implement this you need to specify a prefix, typically `"s"`:

```
sage: W = WeylGroup("A3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: (s1*s2*s1).length()
3
sage: W.long_element()
s1*s2*s3*s1*s2*s1
sage: s1*s2*s3*s1*s2*s1 == s3*s2*s1*s3*s2*s3
True
```

The Weyl group acts on the ambient space of the root lattice, which is accessed by the method `domain`. To illustrate this, recall that if w_0 is the long element then $\alpha \mapsto -w_0(\alpha)$ is a permutation of the simple roots. We may compute this as follows:

```
sage: W = WeylGroup("E6",prefix="s")
sage: w0 = W.long_element(); w0
s1*s3*s4*s5*s6*s2*s4*s5*s3*s4*s1*s3*s2*s4*s5*s6*s2*s4*s5*s3*s4*s1*s3*s2*s4*s5*s3*s4*s1*s3*s2*s4*s1*s
sage: sr = W.domain().simple_roots().list(); sr
[(1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2), (1, 1, 0, 0, 0, 0, 0, 0),
(-1, 1, 0, 0, 0, 0, 0, 0), (0, -1, 1, 0, 0, 0, 0, 0), (0, 0, -1, 1, 0, 0, 0, 0),
(0, 0, 0, -1, 1, 0, 0, 0)]
sage: [-w0.action(a) for a in sr]
[(0, 0, 0, -1, 1, 0, 0, 0), (1, 1, 0, 0, 0, 0, 0, 0), (0, 0, -1, 1, 0, 0, 0, 0),
(0, -1, 1, 0, 0, 0, 0, 0), (-1, 1, 0, 0, 0, 0, 0, 0),
(1/2, -1/2, -1/2, -1/2, -1/2, -1/2, -1/2, 1/2)]
```

We may ask when this permutation is trivial. If it is nontrivial it induces an automorphism of the Dynkin diagram, so it must be nontrivial when the Dynkin diagram has no automorphism. But if there is a nontrivial automorphism, the permutation might or might not be trivial:

```
sage: def roots_not_permuted(ct):
....:     W = WeylGroup(ct)
....:     w0 = W.long_element()
....:     sr = W.domain().simple_roots()
....:     return all(a == -w0.action(a) for a in sr)
....:
sage: for ct in [CartanType(['D',r]) for r in [2..8]]:
....:    print ct,roots_not_permuted(ct)
....:
['D', 2] True
['D', 3] False
```

```
['D', 4] True
['D', 5] False
['D', 6] True
['D', 7] False
['D', 8] True
```

If α is a root let r_α denote the reflection in the hyperplane that is orthogonal to α. We reserve the notation s_α for the simple reflections, that is, the case where α is a simple root. The reflections are just the conjugates of the simple reflections.

The reflections are the keys in a finite family, which is a wrapper around a python dictionary. The values are the positive roots, so given a reflection, you can look up the corresponding root. If you want a list of all reflections, use the method `keys` for the family of reflections:

```
sage: W = WeylGroup("B3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: ref = W.reflections(); ref
Finite family {s1*s2*s1: (1, 0, -1), s2: (0, 1, -1), s3*s2*s3: (0, 1, 1),
s3*s1*s2*s3*s1: (1, 0, 1), s1: (1, -1, 0), s2*s3*s1*s2*s3*s1*s2: (1, 1, 0),
s1*s2*s3*s2*s1: (1, 0, 0), s2*s3*s2: (0, 1, 0), s3: (0, 0, 1)}
sage: ref[s3*s2*s3]
(0, 1, 1)
sage: ref.keys()
[s1*s2*s1, s2, s3*s2*s3, s2*s3*s1*s2*s3*s1*s2, s1, s3*s1*s2*s3*s1, s1*s2*s3*s2*s1, s2*s3*s2, s3]
```

If instead you want a dictionary whose keys are the roots and whose values are the reflections, you may use the inverse family:

```
sage: from pprint import pprint
sage: W = WeylGroup("B3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: altref = W.reflections().inverse_family()
sage: pprint(altref)
Finite family {(1, 0, 0): s1*s2*s3*s2*s1, (0, 1, 1): s3*s2*s3,
               (0, 1, -1): s2, (0, 0, 1): s3, (1, -1, 0): s1,
               (1, 1, 0): s2*s3*s1*s2*s3*s1*s2, (1, 0, -1): s1*s2*s1,
               (1, 0, 1): s3*s1*s2*s3*s1, (0, 1, 0): s2*s3*s2}
sage: [a1,a2,a3] = W.domain().simple_roots()
sage: a1+a2+a3
(1, 0, 0)
sage: altref[a1+a2+a3]
s1*s2*s3*s2*s1
```

The Weyl group is implemented as a GAP matrix group. You therefore can display its character table. The character table is returned as a string, which you can print:

```
sage: print WeylGroup("D4").character_table()
CT1

      2  6  4  5  1  3  5  5  4  3  3  1  4  6
      3  1  .  .  1  .  .  .  .  .  .  1  .  1

        1a 2a 2b 6a 4a 2c 2d 2e 4b 4c 3a 4d 2f

X.1      1  1  1  1  1  1  1  1  1  1  1  1  1
X.2      1 -1  1  1 -1  1  1 -1 -1 -1  1  1  1
X.3      2  .  2 -1  .  2  2  .  .  . -1  2  2
X.4      3 -1 -1  .  1 -1  3 -1  1 -1  . -1  3
```

```
X.5      3 -1 -1  .  1  3 -1 -1 -1  1  . -1  3
X.6      3  1 -1  . -1 -1  3  1 -1  1  . -1  3
X.7      3  1 -1  . -1  3 -1  1  1 -1  . -1  3
X.8      3 -1  3  . -1 -1 -1 -1  1  1  . -1  3
X.9      3  1  3  .  1 -1 -1  1 -1 -1  . -1  3
X.10     4 -2  . -1  .  .  .  2  .  .  1  . -4
X.11     4  2  . -1  .  .  . -2  .  .  1  . -4
X.12     6  . -2  .  . -2 -2  .  .  .  .  2  6
X.13     8  .  .  1  .  .  .  .  .  . -1  . -8
```

Affine Weyl groups

Affine Weyl groups may be created the same way. You simply begin with an affine Cartan type:

```
sage: W = WeylGroup(['A',2,1],prefix="s")
sage: W.cardinality()
+Infinity
sage: [s0,s1,s2] = W.simple_reflections()
sage: s0*s1*s2*s1*s0
s0*s1*s2*s1*s0
```

The affine Weyl group differs from a classical Weyl group since it is infinite. The associated classical Weyl group is a subgroup that may be extracted as follows:

```
sage: W = WeylGroup(['A',2,1],prefix="s")
sage: W1 = W.classical(); W1
Parabolic Subgroup of the Weyl Group of type ['A', 2, 1] (as a matrix group
acting on the root space)
sage: W1.simple_reflections()
Finite family {1: s1, 2: s2}
```

Although `W1` in this example is isomorphic to `WeylGroup("A2")` it has a different matrix realization:

```
sage: for s in WeylGroup(['A',2,1]).classical().simple_reflections():
....:     print s
....:     print
...
[ 1  0  0]
[ 1 -1  1]
[ 0  0  1]

[ 1  0  0]
[ 0  1  0]
[ 1  1 -1]

sage: for s in WeylGroup(['A',2]).simple_reflections():
....:     print s
....:     print
...
[0 1 0]
[1 0 0]
[0 0 1]

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

Bruhat order

The Bruhat partial order on the Weyl group may be defined as follows.

If $u, v \in W$, find a reduced expression of v into a product of simple reflections: $v = s_1 \cdots s_n$. (It is not assumed that the s_i are distinct.) If omitting some of the s_i gives a product that represents u, then $u \leq v$.

The Bruhat order is implemented in Sage as a method of Coxeter groups, and so it is available for Weyl groups, classical or affine.

If $u, v \in W$ then `u.bruhat_le(v)` returns `True` if $u \leq v$ in the Bruhat order.

If $u \leq v$ then the *Bruhat interval* $[u, v]$ is defined to be the set of all t such that $u \leq t \leq v$. One might try to implement this as follows:

```
sage: W = WeylGroup("A2",prefix="s")
sage: [s1,s2] = W.simple_reflections()
sage: def bi(u,v) : return [t for t in W if u.bruhat_le(t) and t.bruhat_le(v)]
...
sage: bi(s1,s1*s2*s1)
[s1*s2*s1, s1*s2, s1, s2*s1]
```

This would not be a good definition since it would fail if W is affine and be inefficient of W is large. Sage has a Bruhat interval method:

```
sage: W = WeylGroup("A2",prefix="s")
sage: [s1,s2] = W.simple_reflections()
sage: W.bruhat_interval(s1,s1*s2*s1)
[s1*s2*s1, s2*s1, s1*s2, s1]
```

This works even for affine Weyl groups.

The Bruhat graph

References:

- [Carrell1994]
- [Deodhar1977]
- [Dyer1993]
- [BumpNakasuji2010]

The *Bruhat graph* is a structure on the Bruhat interval. Suppose that $u \leq v$. The vertices of the graph are x with $u \leq x \leq v$. There is a vertex connecting $x, y \in [x, y]$ if $x = y \cdot r$ where r is a reflection. If this is true then either $x < y$ or $y < x$.

If W is a classical Weyl group the Bruhat graph is implemented in Sage:

```
sage: W = WeylGroup("A3",prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: bg = W.bruhat_graph(s2,s2*s1*s3*s2); bg
Digraph on 10 vertices
sage: bg.show3d()
```

The Bruhat graph has interesting regularity properties that were investigated by Carrell and Peterson. It is a regular graph if both the Kazhdan Lusztig polynomials $P_{u,v}$ and $P_{w_0 v, w_0 u}$ are 1, where w_0 is the long Weyl group element. It is closely related to the *Deodhar conjecture*, which was proved by Deodhar, Carrell and Peterson, Dyer and Polo.

Deodhar proved that if $u < v$ then the Bruhat interval $[u, v]$ contains as many elements of odd length as it does of even length. We observe that often this can be strengthened: If there exists a reflection r such that left (or right) multiplication by r takes the Bruhat interval $[u, v]$ to itself, then this gives an explicit bijection between the elements of odd and even length in $[u, v]$.

Let us search for such reflections. Put the following commands in a file and load the file:

```
W = WeylGroup("A3",prefix="s")
[s1,s2,s3] = W.simple_reflections()
ref = W.reflections().keys()

def find_reflection(u,v):
    bi = W.bruhat_interval(u,v)
    ret = []
    for r in ref:
        if all( r*x in bi for x in bi):
            ret.append(r)
    return ret

for v in W:
    for u in W.bruhat_interval(1,v):
        if u != v:
            print u,v,find_reflection(u,v)
```

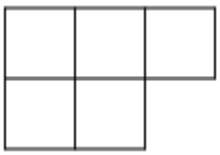
This shows that the Bruhat interval is stabilized by a reflection for all pairs (u, v) with $u < v$ except the following two: $s_3s_1, s_1s_2s_3s_2s_1$ and $s_2, s_2s_3s_1s_2$. Now these are precisely the pairs such that $u \prec v$ in the notation of Kazhdan and Lusztig, and $l(v) - l(u) > 1$. One should not rashly suppose that this is a general characterization of the pairs (u, v) such that no reflection stabilizes the Bruhat interval, for this is not true. However it does suggest that the question is worthy of further investigation.

Classical Crystals

A classical crystal is one coming from the finite (classical) types $A_r, B_r, C_r, D_r, E_{6,7,8}, F_4$, and G_2. Here we describe some background before going into the general theory of crystals and the type dependent combinatorics.

Tableaux and representations of $GL(n)$

Let λ be a partition. The *Young diagram* of λ is the array of boxes having λ_i boxes in the i-th row, left adjusted. Thus if $\lambda = (3, 2)$ the diagram is:



A *semi-standard Young tableau* of shape λ is a filling of the box by integers in which the rows are weakly increasing and the columns are strictly increasing. Thus

1	2	2
2	3	

is a semistandard Young tableau. Sage has a `Tableau` class, and you may create this tableau as follows:

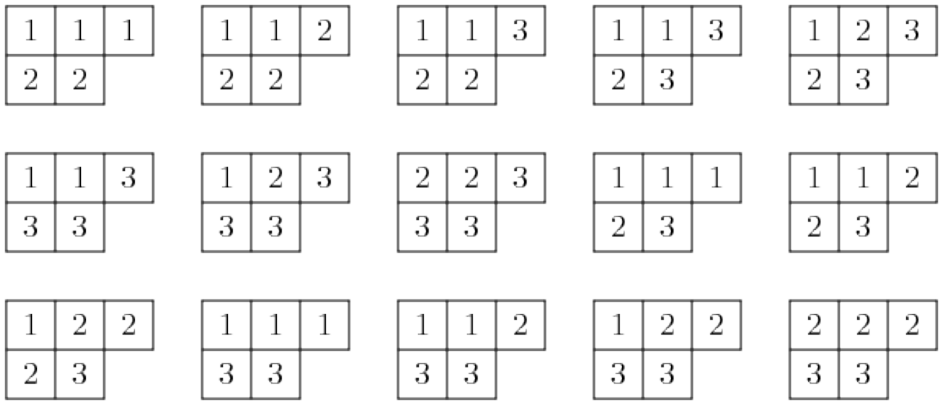
```
sage: T = Tableau([[1,2,2], [2,3]]); T
[[1, 2, 2], [2, 3]]
```

A partition of length $\leq r+1$ is a dominant weight for $GL(r+1, \mathbf{C})$ according to the description of the ambient space in *Standard realizations of the ambient spaces*. Therefore it corresponds to an irreducible representation $\pi_\lambda = \pi_\lambda^{GL(r+1)}$ of $GL(r+1, \mathbf{C})$.

It is true that not every dominant weight λ is a partition, since a dominant weight might have some values λ_i negative. The dominant weight λ is a partition if and only if the character of λ is a polynomial as a function on the space $\mathrm{Mat}_n(\mathbf{C})$. Thus for example $\det^{-1} = \pi_\lambda$ with $\lambda = (-1, \ldots, -1)$, which is a dominant weight but not a partition, and the character is not a polynomial function on $\mathrm{Mat}_n(\mathbf{C})$.

Theorem [Littlewood] If λ is a partition, then the number of semi-standard Young tableaux with shape λ and entries in $\{1, 2, \ldots, r+1\}$ is the dimension of π_λ.

For example, if $\lambda = (3, 2)$ and $r = 2$, then we find 15 tableaux with shape λ and entries in $\{1, 2, 3\}$:



This is consistent with the theorem since the dimension of the irreducible representation of $GL(3)$ with highest weight $(3, 2, 0)$ has dimension 15:

```
sage: A2 = WeylCharacterRing("A2")
sage: A2(3,2,0).degree()
15
```

In fact we may obtain the character of the representation from the set of tableaux. Indeed, one of the definitions of the Schur polynomial (due to Littlewood) is the following combinatorial one. If T is a tableaux, define the *weight* of T to be $\mathrm{wt}(T) = (k_1, \ldots, k_n)$ where k_i is the number of i's in the tableaux. Then the multiplicity of μ in the character χ_λ is the number of tableaux of weight λ. Thus if $\mathbf{z} = (z_1, \ldots, z_n)$, we have

$$\chi_\lambda(\mathbf{z}) = \sum_T \mathbf{z}^{\mathrm{wt}(T)}$$

where the sum is over all semi-standard Young tableaux of shape λ that have entries in $\{1, 2, \ldots, r+1\}$.

Frobenius-Schur Duality

Frobenius-Schur duality is a relationship between the representation theories of the symmetric group and general linear group. We will relate this to tableaux in the next section.

Representations of the symmetric group S_k are parametrized by partitions λ of k. The parametrization may be characterized as follows. Let n be any integer $\geq k$. Then both $GL(n, \mathbf{C})$ and S_k act on $\otimes^k V$ where $V = \mathbf{C}^n$. Indeed, $GL(n)$ acts on each V and S_k permutes them. Then if $\pi_\lambda^{GL(n)}$ is the representation of $GL(n, \mathbf{C})$ with highest weight

vector λ and $\pi_\lambda^{S_k}$ is the irreducible representation of S_k parametrized by λ then

$$\otimes^k V \cong \bigoplus_{\lambda \vdash k} \pi_\lambda^{GL(n)} \otimes \pi_\lambda^{S_k}$$

as bimodules for the two groups. This is *Frobenius-Schur duality* and it serves to characterize the parametrization of the irreducible representations of S_k by partitions of k.

Counting pairs of tableaux

In both the representation theory of $GL(n)$ and the representation theory of S_k, the degrees of irreducible representations can be expressed in terms of the number of tableaux of the appropriate type. We have already stated the theorem for $GL(n)$. For S_k, it goes as follows.

Let us say that a semistandard Young tableau T of shape $\lambda \vdash k$ is *standard* if T contains each entry $1, 2, \ldots, k$ exactly once. Thus both rows and columns are strictly increasing.

Theorem [Young, 1927] The degree of π_λ is the number of standard tableaux of shape λ.

Now let us consider the implications of Frobenius-Schur duality. The dimension of $\otimes^k V$ is n^k. Therefore n^k is equal to the number of pairs $(T1, T2)$ of tableaux of the same shape $\lambda \vdash k$, where the first tableaux is standard (in the alphabet $1, 2, \ldots, k$), and the second the second semistandard (in the alphabet $1, 2, \ldots, n$).

The Robinson-Schensted-Knuth correspondence

The last purely combinatorial statement has a combinatorial proof, based on the Robinson-Schensted-Knuth (RSK) correspondence.

References:

- [Knuth1998], section "Tableaux and Involutions".
- [Knuth1970]
- [Fulton1997]
- [Stanley1999]

The RSK correspondence gives bijections between pairs of tableaux of various types and combinatorial objects of different types. We will not review the correspondence in detail here, but see the references. We note that Schensted insertion is implemented as the method `schensted_insertion` of Tableau class in Sage.

Thus we have the following bijections:

- Pairs of tableaux T_1 and T_2 of shape λ where λ runs through the partitions of k such that T_1 is a standard tableau and T_2 is a semistandard tableau in $1, 2, \ldots, n$ are in bijection with the n^k words of length k in $1, 2, \ldots, n$.
- Pairs of standard tableaux of the same shape λ as λ runs through the partitions of k are in bijection with the $k!$ elements of S_k.
- Pairs of tableaux T_1 and T_2 of the same shape λ but arbitrary size in $1, 2, 3, \ldots, n$ are in bijection with $n \times n$ positive integer matrices.
- Pairs of tableaux T_1 and T_2 of conjugate shapes λ and λ' are in bijection with $n \times n$ matrices with entries 0 or 1.

The second of these four bijection gives a combinatorial proof of the fact explained above, that the number of pairs $(T1, T2)$ of tableaux of the same shape $\lambda \vdash k$, where the first tableaux is standard (in the alphabet $1, 2, \cdots, k$), and the second the second semistandard (in the alphabet $1, 2, \cdots, n$). So this second bijection is a *combinatorial analog of Frobenius-Schur duality*.

Analogies between representation theory and combinatorics

The four combinatorial bijections (variants of RSK) cited above have the following analogs in representation theory.

- The first combinatorial fact corresponds to Frobenius-Schur duality, as we have already explained.
- The second combinatorial fact also has an analog in representation theory. The group algebra $\mathbf{C}[S_k]$ is an $S_k \times S_k$ bimodule with of dimension $k!$. It decomposes as a direct sum of $\pi_\lambda^{S_k} \otimes \pi_\lambda^{S_k}$.

Both the combinatorial fact and the decomposition of $\mathbf{C}[S_k]$ show that the number of pairs of standard tableaux of size k and the same shape equals $k!$.

- The third combinatorial fact is analogous to the decomposition of the ring of polynomial functions on $\mathrm{Mat}(n, \mathbf{C})$ on which $GL(n, \mathbf{C}) \times GL(n, \mathbf{C})$ acts by $(g_1, g_2)f(X) = f({}^t g_1 X g_2)$. The polynomial ring decomposes into the direct sum of $\pi_\lambda^{GL(n)} \otimes \pi_\lambda^{GL(n)}$.

Taking traces gives the *Cauchy identity*:

$$\sum_\lambda s_\lambda(x_1, \cdots, x_n) s_\lambda(y_1, \cdots, y_n) = \prod_{i,j} (1 - x_i y_j)^{-1}$$

where x_i are the eigenvalues of g_1 and y_j are the eigenvalues of g_2. The sum is over all partitions λ.

- The last combinatorial fact is analogous to the decomposition of the exterior algebra over $\mathrm{Mat}(n, \mathbf{C})$.

Taking traces gives the *dual Cauchy identity*:

$$\sum_\lambda s_\lambda(x_1, \cdots, x_n) s_{\lambda'}(y_1, \cdots, y_n) = \prod_{i,j} (1 + x_i y_j).$$

Again the sum is over partitions λ and here λ' is the conjugate partition.

Interpolating between representation theory and combinatorics

The theory of quantum groups interpolates between the representation theoretic picture and the combinatorial picture, and thereby explains these analogies. The representation $\pi_\lambda^{GL(n)}$ is reinterpreted as a module for the quantized enveloping algebra $U_q(\mathfrak{gl}_n(\mathbf{C}))$, and the representation $\pi_\lambda^{S_k}$ is reinterpreted as a module for the Iwahori Hecke algebra. Then Frobenius-Schur duality persists. See [Jimbo1986]. When $q \to 1$, we recover the representation story. When $q \to 0$, we recover the combinatorial story.

Kashiwara crystals

References:

- [Kashiwara1995]
- [KashiwaraNakashima1994]
- [HongKang2002]

Kashiwara considered the highest weight modules of quantized enveloping algebras $U_q(\mathfrak{g})$ in the limit when $q \to 0$. The enveloping algebra cannot be defined when $q = 0$, but a limiting structure can still be detected. This is the *crystal basis* of the module.

Kashiwara's crystal bases have a combinatorial structure that sheds light even on purely combinatorial constructions on tableaux that predated quantum groups. It gives a good generalization to other Cartan types (or more generally to Kac-Moody algebras).

Let Λ be the weight lattice of a Cartan type with root system Φ. We now define a *crystal* of type Φ. Let $\mathcal{B}$ be a set, and let $0 \notin \mathcal{B}$ be an auxiliary element. For each index $1 \leq i \leq r$ we assume there given maps $e_i, f_i : \mathcal{B} \longrightarrow \mathcal{B} \cup \{0\}$, maps $\varepsilon_i, \varphi_i : \mathcal{B} \longrightarrow \mathbf{Z}$ and a map $\mathrm{wt} : \mathcal{B} \longrightarrow \Lambda$ satisfying certain assumptions, which we now describe. It is assumed that if $x, y \in \mathcal{B}$ then $e_i(x) = y$ if and only if $f_i(y) = x$. In this case, it is assumed that

$$\mathrm{wt}(y) = \mathrm{wt}(x) + \alpha_i, \qquad \varepsilon_i(x) = \varepsilon_i(y) + 1, \qquad \varphi_i(x) = \varphi_i(y) - 1.$$

Moreover, we assume that

$$\varphi_i(x) - \varepsilon_i(x) = \langle \mathrm{wt}(x), \alpha_i^\vee \rangle$$

for all $x \in \mathcal{B}$.

We call a crystal *regular* if it satisfies the additional assumption that $\varepsilon_i(v)$ is the number of times that e_i may be applied to v, and that $\phi_i(v)$ is the number of times that f_i may be applied. That is, $\varphi_i(x) = \max\{k | f_i^k x \neq 0\}$ and $\varepsilon_i(x) = \max\{k | e_i^k(x) \neq 0\}$. Kashiwara also allows ε_i and φ_i to take the value $-\infty$.

Note: Most of the crystals that we are concerned with here are regular.

Given the crystal $\mathcal{B}$, the *character* $\chi_{\mathcal{B}}$ is:

$$\sum_{v \in \mathcal{B}} \mathbf{z}^{wt(v)}.$$

Given any highest weight λ, constructions of Kashiwara and Nakashima, Littelmann and others produce a crystal $\chi_{\mathcal{B}_\lambda}$ such that $\chi_{\mathcal{B}_\lambda} = \chi_\lambda$, where χ_λ is the irreducible character with highest weight λ, as in *Representations and characters*.

The crystal $\mathcal{B}_\lambda$ is not uniquely characterized by the properties that we have stated so far. For Cartan types A, D, E (more generally, any simply-laced type) it may be characterized by these properties together with certain other *Stembridge axioms*. We will take it for granted that there is a unique "correct" crystal $\mathcal{B}_\lambda$ and discuss how these are constructed in Sage.

Installing dot2tex

Before giving examples of crystals, we digress to help you install `dot2tex`, which you will need in order to make latex images of crystals.

`dot2tex` is an optional package of sage and the latest version can be installed via:

```
sage -i dot2tex
```

Crystals of tableaux in Sage

All crystals that are currently in Sage can be accessed by `crystals.<tab>`.

For type A_r, Kashiwara and Nakashima put a crystal structure on the set of tableaux with shape λ in $1, 2, \ldots, r+1$, and this is a realization of $\mathcal{B}_\lambda$. Moreover, this construction extends to other Cartan types, as we will explain. At the moment, we will consider how to draw pictures of these crystals.

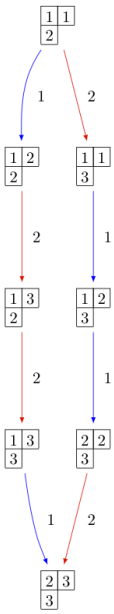
Once you have `dot2tex` installed, you may make images pictures of crystals with a command such as this:

```
sage: crystals.Tableaux("A2", shape=[2,1]).latex_file("/tmp/a2rho.tex") # optional - dot2tex graphvi
```

Here $\lambda = (2,1) = (2,1,0)$. The crystal `C` is $\mathcal{B}_\lambda$. The character χ_λ will therefore be the eight-dimensional irreducible character with this highest weight. Then you may run pdflatex on the file `a2rho.tex`. This can also be achieved without the detour of saving the latex file via:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: view(B, pdflatex=True, tightpage=True) # optional - dot2tex graphviz
```

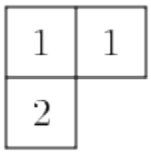
This produces the crystal graph:



You may also wish to color the edges in different colors by specifying further latex options:

```
sage: B = crystals.Tableaux(['A',2], shape=[2,1])
sage: G = B.digraph()
sage: G.set_latex_options(color_by_label = {1:"red", 2:"yellow"})
sage: view(G, pdflatex=True, tightpage=True) # optional - dot2tex graphviz
```

As you can see, the elements of this crystal are exactly the eight tableaux of shape λ with entries in $\{1,2,3\}$. The convention is that if $x, y \in \mathcal{B}$ and $f_i(x) = y$, or equivalently $e_i(y) = x$, then we draw an arrow from $x \to y$. Thus the highest weight tableau is the one with no incoming arrows. Indeed, this is:



We recall that the weight of the tableau is (k_1, k_2, k_3) where k_i is the number of i's in the tableau, so this tableau has weight $(2, 1, 0)$, which indeed equals λ.

Once the crystal is created, you have access to the ambient space and its methods through the method `weight_lattice_realization()`:

```
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: L = C.weight_lattice_realization(); L
Ambient space of the Root system of type ['A', 2]
sage: L.fundamental_weights()
Finite family {1: (1, 0, 0), 2: (1, 1, 0)}
```

The highest weight vector is available as follows:

```
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: v = C.highest_weight_vector(); v
[[1, 1], [2]]
```

or more simply:

```
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: C[0]
[[1, 1], [2]]
```

Now we may apply the operators e_i and f_i to move around in the crystal:

```
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: v = C.highest_weight_vector()
sage: v.f(1)
[[1, 2], [2]]
sage: v.f(1).f(1)
sage: v.f(1).f(1) is None
True
sage: v.f(1).f(2)
[[1, 3], [2]]
sage: v.f(1).f(2).f(2)
[[1, 3], [3]]
sage: v.f(1).f(2).f(2).f(1)
[[2, 3], [3]]
sage: v.f(1).f(2).f(2).f(1) == v.f(2).f(1).f(1).f(2)
True
```

You can construct the character if you first make a Weyl character ring:

```
sage: A2 = WeylCharacterRing("A2")
sage: C = crystals.Tableaux("A2", shape=[2,1])
sage: C.character(A2)
A2(2,1,0)
```

Crystals of letters

For each of the classical Cartan types there is a *standard crystal* $\mathcal{B}_{\text{standard}}$ from which other crystals can be built up by taking tensor products and extracting constituent irreducible crystals. This procedure is sufficient for Cartan types A_r and C_r. For types B_r and D_r the standard crystal must be supplemented with *spin crystals*. See [KashiwaraNakashima1994] or [HongKang2002] for further details.

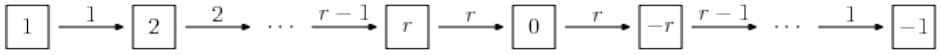
Here is the standard crystal of type A_r.

You may create the crystal and work with it as follows:

```
sage: C = crystals.Letters("A6")
sage: v0 = C.highest_weight_vector(); v0
1
sage: v0.f(1)
2
sage: v0.f(1).f(2)
3
sage: [v0.f(1).f(2).f(x) for x in [1..6]]
```

```
[None, None, 4, None, None, None]
sage: [v0.f(1).f(2).e(x) for x in [1..6]]
[None, 2, None, None, None, None]
```

Here is the standard crystal of type B_r.



There is, additionally, a spin crystal for B_r, corresponding to the 2^r-dimensional spin representation. We will not draw it, but we will describe it. Its elements are vectors $\epsilon_1 \otimes \cdots \otimes \epsilon_r$, where each `spin` $\epsilon_i = \pm$.

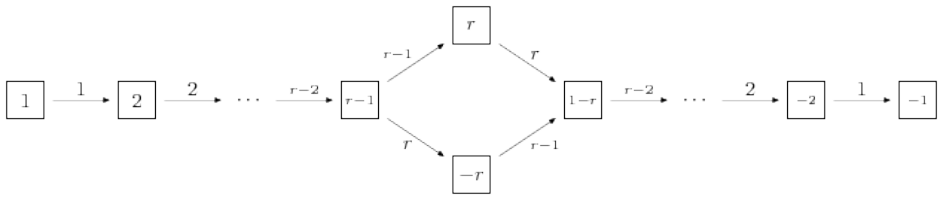
If $i < r$, then the effect of the operator f_i is to annihilate $v = \epsilon_1 \otimes \cdots \otimes \epsilon_r$ unless $\epsilon_i \otimes \epsilon_{i+1} = + \otimes -$. If this is so, then $f_i(v)$ is obtained from v by replacing $\epsilon_i \otimes \epsilon_{i+1}$ by $- \otimes +$. If $i = r$, then f_r annihilates v unless $\epsilon_r = +$, in which case it replaces ϵ_r by $-$.

Create the spin crystal as follows. The crystal elements are represented in the signature representation listing the ϵ_i:

```
sage: C = crystals.Spins("B3")
sage: C.list()
[+++, ++-, +-+, -++, +--, -+-, --+, ---]
```

Here is the standard crystal of type C_r.

Here is the standard crystal of type D_r.



There are two spin crystals for type D_r. Each consists of $\epsilon_1 \otimes \cdots \otimes \epsilon_r$ with $\epsilon_i = \pm$, and the number of spins either always even or always odd. We will not describe the effect of the root operators f_i, but you are invited to create them and play around with them to guess the rule:

```
sage: Cplus = crystals.SpinsPlus("D4")
sage: Cminus = crystals.SpinsMinus("D4")
```

It is also possible to construct the standard crystal for type G_2, E_6, and E_7. Here is the one for type G_2 (corresponding to the representation of degree 7):

The crystal of letters is a special case of the crystal of tableaux in the sense that $\mathcal{B}_{\text{standard}}$ is isomorphic to the crystal of tableaux whose highest weight λ is the highest weight vector of the standard representation. Thus compare:

```
sage: crystals.Letters("A3")
The crystal of letters for type ['A', 3]
sage: crystals.Tableaux("A3", shape=[1])
The crystal of tableaux of type ['A', 3] and shape(s) [[1]]
```

These two crystals are different in implementation, but they are isomorphic. In fact the second crystal is constructed from the first. We can test isomorphisms between crystals as follows:

```
sage: Cletter = crystals.Letters(['A',3])
sage: Ctableaux = crystals.Tableaux(['A',3], shape = [1])
sage: Cletter.digraph().is_isomorphic(Ctableaux.digraph())
True
sage: Cletter.digraph().is_isomorphic(Ctableaux.digraph(), certify = True)
(True, {1: [[1]], 2: [[2]], 3: [[3]], 4: [[4]]})
```

where in the last step the explicit map between the vertices of the crystals is given.

Crystals of letters have a special role in the theory since they are particularly simple, yet as Kashiwara and Nakashima showed, the crystals of tableaux can be created from them. We will review how this works.

Tensor products of crystals

Kashiwara defined the tensor product of crystals in a purely combinatorial way. The beauty of this construction is that it exactly parallels the tensor product of crystals of representations. That is, if λ and μ are dominant weights, then $\mathcal{B}_\lambda \otimes \mathcal{B}_\mu$ is a (usually disconnected) crystal, which may contain multiple copies of $\mathcal{B}_\nu$ (for another dominant weight ν), and the number of copies of $\mathcal{B}_\nu$ is exactly the multiplicity of χ_ν in $\chi_\lambda \chi_\mu$.

We will describe two conventions for the tensor product of crystals.

Kashiwara's definition As a set, the tensor product $\mathcal{B} \otimes \mathcal{C}$ of crystals $\mathcal{B}$ and $\mathcal{C}$ is the Cartesian product, but we denote the ordered pair (x, y) with $x \in \mathcal{B}$ and $y \in \mathcal{C}$ by $x \otimes y$. We define $\mathrm{wt}(x \otimes y) = \mathrm{wt}(x) + \mathrm{wt}(y)$. We define

$$f_i(x \otimes y) = \begin{cases} f_i(x) \otimes y & \text{if } \varphi_i(x) > \varepsilon_i(y), \\ x \otimes f_i(y) & \text{if } \varphi_i(x) \leq \varepsilon_i(y), \end{cases}$$

and

$$e_i(x \otimes y) = \begin{cases} e_i(x) \otimes y & \text{if } \varphi_i(x) \geq \varepsilon_i(y), \\ x \otimes e_i(y) & \text{if } \varphi_i(x) < \varepsilon_i(y). \end{cases}$$

It is understood that $x \otimes 0 = 0 \otimes x = 0$. We also define:

$$\varphi_i(x \otimes y) = \max(\varphi_i(y), \varphi_i(x) + \varphi_i(y) - \varepsilon_i(y)),$$

$$\varepsilon_i(x \otimes y) = \max(\varepsilon_i(x), \varepsilon_i(x) + \varepsilon_i(y) - \varphi_i(x)).$$

Alternative definition As a set, the tensor product $\mathcal{B} \otimes \mathcal{C}$ of crystals $\mathcal{B}$ and $\mathcal{C}$ is the Cartesian product, but we denote the ordered pair (y, x) with $y \in \mathcal{B}$ and $x \in \mathcal{C}$ by $x \otimes y$. We define $\mathrm{wt}(x \otimes y) = \mathrm{wt}(y) + \mathrm{wt}(x)$. We define

$$f_i(x \otimes y) = \begin{cases} f_i(x) \otimes y & \text{if } \varphi_i(y) \leq \varepsilon_i(x), \\ x \otimes f_i(y) & \text{if } \varphi_i(y) > \varepsilon_i(x), \end{cases}$$

and

$$e_i(x \otimes y) = \begin{cases} e_i(x) \otimes y & \text{if } \varphi_i(y) < \varepsilon_i(x), \\ x \otimes e_i(y) & \text{if } \varphi_i(y) \geq \varepsilon_i(x). \end{cases}$$

It is understood that $y \otimes 0 = 0 \otimes y = 0$. We also define

$$\varphi_i(x \otimes y) = \max(\varphi_i(x), \varphi_i(y) + \varphi_i(x) - \varepsilon_i(x)),$$

$$\varepsilon_i(x \otimes y) = \max(\varepsilon_i(y), \varepsilon_i(y) + \varepsilon_i(x) - \varphi_i(y)).$$

The tensor product is associative: $(x \otimes y) \otimes z \mapsto x \otimes (y \otimes z)$ is an isomorphism $(\mathcal{B} \otimes \mathcal{C}) \otimes \mathcal{D} \to \mathcal{B} \otimes (\mathcal{C} \otimes \mathcal{D})$, and so we may consider tensor products of arbitrary numbers of crystals.

The relationship between the two definitions The relationship between the two definitions is simply that the Kashiwara tensor product $\mathcal{B} \otimes \mathcal{C}$ is the alternate tensor product $\mathcal{C} \otimes \mathcal{B}$ in reverse order. Sage uses the alternative tensor product. Even though the tensor product construction is *a priori* asymmetrical, both constructions produce isomorphic crystals, and in particular Sage's crystals of tableaux are identical to Kashiwara's.

Note: Using abstract crystals (i.e. they satisfy the axioms but do not arise from a representation of $U_q(\mathfrak{g})$), we can construct crystals $\mathcal{B}, \mathcal{C}$ such that $\mathcal{B} \otimes \mathcal{C} \neq \mathcal{C} \otimes \mathcal{B}$ (of course, using the same convention).

Tensor products of crystals in Sage You may construct the tensor product of several crystals in Sage using `crystals.TensorProduct`:

```
sage: C = crystals.Letters("A2")
sage: T = crystals.TensorProduct(C,C,C); T
Full tensor product of the crystals [The crystal of letters for type ['A', 2],
The crystal of letters for type ['A', 2], The crystal of letters for type ['A', 2]]
sage: T.cardinality()
27
sage: T.highest_weight_vectors()
[[1, 1, 1], [1, 2, 1], [2, 1, 1], [3, 2, 1]]
```

This crystal has four highest weight vectors. We may understand this as follows:

```
sage: A2 = WeylCharacterRing("A2")
sage: C = crystals.Letters("A2")
sage: T = crystals.TensorProduct(C,C,C)
sage: chi_C = C.character(A2)
sage: chi_T = T.character(A2)
sage: chi_C
A2(1,0,0)
sage: chi_T
A2(1,1,1) + 2*A2(2,1,0) + A2(3,0,0)
sage: chi_T == chi_C^3
True
```

As expected, the character of `T` is the cube of the character of `C`, and representations with highest weight $(1,1,1)$, $(3,0,0)$ and $(2,1,0)$. This decomposition is predicted by Frobenius-Schur duality: the multiplicity of $\pi_\lambda^{GL(n)}$ in $\otimes^3 \mathbf{C}^3$ is the degree of $\pi_\lambda^{S_3}$.

It is useful to be able to select one irreducible constitutent of `T`. If we only want one of the irreducible constituents of `T`, we can specify a list of highest weight vectors by the option `generators`. If the list has only one element, then we get an irreducible crystal. We can make four such crystals:

```
sage: A2 = WeylCharacterRing("A2")
sage: C = crystals.Letters("A2")
sage: T = crystals.TensorProduct(C,C,C)
sage: [T1,T2,T3,T4] = \
  [crystals.TensorProduct(C,C,C,generators=[v]) for v in T.highest_weight_vectors()]
sage: [B.cardinality() for B in [T1,T2,T3,T4]]
[10, 8, 8, 1]
sage: [B.character(A2) for B in [T1,T2,T3,T4]]
[A2(3,0,0), A2(2,1,0), A2(2,1,0), A2(1,1,1)]
```

We see that two of these crystals are isomorphic, with character `A2(2,1,0)`. Try:

```
sage: A2 = WeylCharacterRing("A2")
sage: C = crystals.Letters("A2")
```

```
sage: T = crystals.TensorProduct(C,C,C)
sage: [T1,T2,T3,T4] = \
  [crystals.TensorProduct(C,C,C,generators=[v]) for v in T.highest_weight_vectors()]
sage: T1.plot()
sage: T2.plot()
sage: T3.plot()
sage: T4.plot()
```

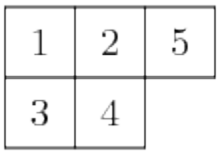
Elements of `crystals.TensorProduct(A,B,C, ...)` are represented by sequences `[a,b,c, ...]` with `a` in `A`, `b` in `B`, etc. This of course represents $a \otimes b \otimes c \otimes \cdots$.

Crystals of tableaux as tensor products of crystals

Sage implements the `CrystalOfTableaux` as a subcrystal of a tensor product of the `ClassicalCrystalOfLetters`. You can see how its done as follows:

```
sage: T = crystals.Tableaux("A4",shape=[3,2])
sage: v = T.highest_weight_vector().f(1).f(2).f(3).f(2).f(1).f(4).f(2).f(3); v
[[1, 2, 5], [3, 4]]
sage: v._list
[3, 1, 4, 2, 5]
```

We've looked at the internal representation of v, where it is represented as an element of the fourth tensor power of the `ClassicalCrystalOfLetters`. We see that the tableau:



is interpreted as the tensor:

The elements of the tableau are read from bottom to top and from left to right. This is the *inverse middle-Eastern reading* of the tableau. See Hong and Kang, *loc. cit.* for discussion of the readings of a tableau.

Spin crystals

For the Cartan types A_r, C_r or G_2, `CrystalOfTableaux` are capable of making any finite crystal. (For type A_r it is necessary that the highest weight λ be a partition.)

For Cartan types B_r and D_r, there also exist spin representations. The corresponding crystals are implemented as *spin crystals*. For these types, `CrystalOfTableaux` also allows the input shape λ to be half-integral if it is of height r. For example:

```
sage: C = crystals.Tableaux(['B',2], shape = [3/2, 1/2])
sage: C.list()
[[++, [[1]]], [++, [[2]]], [++, [[0]]], [++, [[-2]]], [++, [[-1]]], [+-, [[-2]]],
```

```
[+-, [[-1]]], [-+, [[-1]]], [+-, [[1]]], [+-, [[2]]], [-+, [[2]]], [+-, [[0]]],
[-+, [[0]]], [-+, [[-2]]], [--, [[-2]]], [--, [[-1]]]]
```

Here the first list of $+$ and $-$ gives a spin column that is discussed in more detail in the next section and the second entry is a crystal of tableau element for $\lambda = (\lfloor\lambda_1\rfloor, \lfloor\lambda_2\rfloor, \ldots)$. For type D_r, we have the additional feature that there are two types of spin crystals. Hence in `CrystalOfTableaux` the r-th entry of λ in this case can also take negative values:

```
sage: C = crystals.Tableaux(['D',3], shape = [1/2, 1/2,-1/2])
sage: C.list()
[[++-, []], [+-+, []], [-++, []], [---, []]]
```

For rank two Cartan types, we also have `crystals.FastRankTwo` which gives a different fast implementation of these crystals:

```
sage: B = crystals.FastRankTwo(['B',2], shape=[3/2,1/2]); B
The fast crystal for B2 with shape [3/2,1/2]
sage: v = B.highest_weight_vector(); v.weight()
(3/2, 1/2)
```

Type B spin crystal The spin crystal has highest weight $(1/2, \ldots, 1/2)$. This is the last fundamental weight. The irreducible representation with this weight is the spin representation of degree 2^r. Its crystal is hand-coded in Sage:

```
sage: Cspin = crystals.Spins("B3"); Cspin
The crystal of spins for type ['B', 3]
sage: Cspin.cardinality()
8
```

The crystals with highest weight λ, where λ is a half-integral weight, are constructed as a tensor product of a spin column and the highest weight crystal of the integer part of λ. For example, suppose that $\lambda = (3/2, 3/2, 1/2)$. The corresponding irreducible character will have degree 112:

```
sage: B3 = WeylCharacterRing("B3")
sage: B3(3/2,3/2,1/2).degree()
112
```

So $\mathcal{B}_\lambda$ will have 112 elements. We can find it as a subcrystal of `Cspin` $\otimes \mathcal{B}_\mu$, where $\mu = \lambda - (1/2, 1/2, 1/2) = (1, 1, 0)$:

```
sage: B3 = WeylCharacterRing("B3")
sage: B3(1,1,0)*B3(1/2,1/2,1/2)
B3(1/2,1/2,1/2) + B3(3/2,1/2,1/2) + B3(3/2,3/2,1/2)
```

We see that just taking the tensor product of these two crystals will produce a reducible crystal with three constitutents, and we want to extract the one we want. We do that as follows:

```
sage: B3 = WeylCharacterRing("B3")
sage: C1 = crystals.Tableaux("B3", shape=[1,1])
sage: Cspin = crystals.Spins("B3")
sage: C = crystals.TensorProduct(C1, Cspin, generators=[[C1[0],Cspin[0]]])
sage: C.cardinality()
112
```

Alternatively, we can get this directly from `CrystalOfTableaux`:

```
sage: C = crystals.Tableaux(['B',3], shape = [3/2,3/2,1/2])
sage: C.cardinality()
112
```

This is the desired crystal.

Type D spin crystals A similar situation pertains for type D_r, but now there are two spin crystals, both of degree 2^{r-1}. These are hand-coded in sage:

```
sage: SpinPlus = crystals.SpinsPlus("D4")
sage: SpinMinus = crystals.SpinsMinus("D4")
sage: SpinPlus[0].weight()
(1/2, 1/2, 1/2, 1/2)
sage: SpinMinus[0].weight()
(1/2, 1/2, 1/2, -1/2)
sage: [C.cardinality() for C in [SpinPlus,SpinMinus]]
[8, 8]
```

Similarly to type B crystal, we obtain crystal with spin weight by allowing for partitions with half-integer values, and the last entry can be negative depending on the type of the spin.

Lusztig involution

The Lusztig involution on a finite-dimensional highest weight crystal $B(\lambda)$ of highest weight λ maps the highest weight vector to the lowest weight vector and the Kashiwara operator f_i to e_{i^*}, where i^* is defined as $\alpha_{i^*} = -w_0(\alpha_i)$. Here w_0 is the longest element of the Weyl group acting on the i-th simple root α_i. For example, for type A_n we have $i^* = n+1-i$, whereas for type C_n we have $i^* = i$. For type D_n and n even also have $i^* = i$, but for n odd this map interchanges nodes $n-1$ and n. Here is how to achieve this in Sage:

```
sage: B = crystals.Tableaux(['A',3],shape=[2,1])
sage: b = B(rows=[[1,2],[3]])
sage: b.lusztig_involution()
[[2, 4], [3]]
```

For type A_n, the Lusztig involution is the same as the Schuetzenberger involution (which in Sage is defined on tableaux):

```
sage: t = Tableau([[1,2],[3]])
sage: t.schuetzenberger_involution(n=4)
[[2, 4], [3]]
```

For all tableaux in a given crystal, this can be tested via:

```
sage: B = crystals.Tableaux(['A',3],shape=[2])
sage: all(b.lusztig_involution().to_tableau() == b.to_tableau().schuetzenberger_involution(n=4) for b
True
```

The Lusztig involution is also defined for finite-dimensional highest weight crystals of exceptional type:

```
sage: C = CartanType(['E',6])
sage: La = C.root_system().weight_lattice().fundamental_weights()
sage: T = crystals.HighestWeight(La[1])
sage: t = T[4]; t
[(-2, 5)]
sage: t.lusztig_involution()
[(-3, 2)]
```

Levi branching rules for crystals

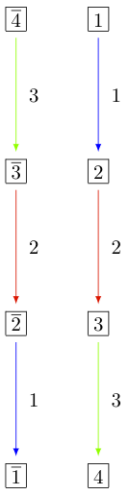
Let G be a Lie group and H a Levi subgroup. We have already seen that the Dynkin diagram of H is obtained from that of G by erasing one or more nodes.

If $\mathcal{C}$ is a crystal for G, then we may obtain the corresponding crystal for H by a similar process. For example if the Dynkin diagram for H is obtained from the Dynkin diagram for G by erasing the i-th node, then if we erase all the edges in the crystal $\mathcal{C}$ that are labeled with i, we obtain a crystal for H.

In Sage this is achieved by specifying the index set used in the digraph method:

```
sage: T = crystals.Tableaux(['D',4],shape=[1])
sage: G = T.digraph(index_set=[1,2,3])
```

We see that the type D_4 crystal indeed decomposes into two type A_3 components.



For more on branching rules, see *Maximal Subgroups and Branching Rules* or *Levi subgroups* for specifics on the Levi subgroups.

Subcrystals

Sometimes it might be desirable to work with a subcrystal of a crystal. For example, one might want to look at all $\{2, 3, \ldots, n\}$ highest elements of a crystal and look at a particular such component:

```
sage: T = crystals.Tableaux(['D',4],shape=[2,1])
sage: hw = [ t for t in T if t.is_highest_weight(index_set = [2,3,4]) ]; hw
[[[1, 1], [2]],
 [[1, 2], [2]],
 [[2, -1], [-2]],
 [[2, -1], [-1]],
 [[1, -1], [2]],
 [[2, -1], [3]],
 [[1, 2], [3]],
 [[2, 2], [3]],
 [[1, 2], [-2]],
 [[2, 2], [-2]],
 [[2, 2], [-1]]]
sage: C = T.subcrystal(generators = [T(rows=[[2,-1],[3]])], index_set = [2,3,4])
sage: G = T.digraph(subset = C, index_set=[2,3,4])
```

Affine Finite Crystals

In this document we briefly explain the construction and implementation of the Kirillov–Reshetikhin crystals of [FourierEtAl2009].

Kirillov–Reshetikhin (KR) crystals are finite-dimensional affine crystals corresponding to Kirillov–Reshektikhin modules. They were first conjectured to exist in [HatayamaEtAl2001]. The proof of their existence for nonexceptional types was given in [OkadoSchilling2008] and their combinatorial models were constructed in [FourierEtAl2009]. Kirillov-Reshetikhin crystals $B^{r,s}$ are indexed first by their type (like $A_n^{(1)}$, $B_n^{(1)}$, ...) with underlying index set $I = \{0, 1, \ldots, n\}$ and two integers r and s. The integers s only needs to satisfy $s > 0$, whereas r is a node of the finite Dynkin diagram $r \in I \setminus \{0\}$.

Their construction relies on several cases which we discuss separately. In all cases when removing the zero arrows, the crystal decomposes as a (direct sum of) classical crystals which gives the crystal structure for the index set $I_0 = \{1, 2, \ldots, n\}$. Then the zero arrows are added by either exploiting a symmetry of the Dynkin diagram or by using embeddings of crystals.

Type $A_n^{(1)}$

The Dynkin diagram for affine type A has a rotational symmetry mapping $\sigma : i \mapsto i + 1$ where we view the indices modulo $n + 1$:

```
sage: C = CartanType(['A',3,1])
sage: C.dynkin_diagram()
0
O--------+
|        |
|        |
O---O---O
1   2   3
A3~
```

The classical decomposition of $B^{r,s}$ is the A_n highest weight crystal $B(s\omega_r)$ or equivalently the crystal of tableaux labelled by the rectangular partition (s^r):

$$B^{r,s} \cong B(s\omega_r) \quad \text{as a } \{1, 2, \ldots, n\}\text{-crystal}$$

In Sage we can see this via:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1],1,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['A', 3] and shape(s) [[1]]
sage: K.list()
[[[1]], [[2]], [[3]], [[4]]]

sage: K = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['A', 3] and shape(s) [[1, 1]]
```

One can change between the classical and affine crystal using the methods `lift` and `retract`:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: b = K(rows=[[1],[3]]); type(b)
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_A_with_category.element_class'>
sage: b.lift()
[[1], [3]]
sage: type(b.lift())
```

```
<class 'sage.combinat.crystals.tensor_product.CrystalOfTableaux_with_category.element_class'>

sage: b = crystals.Tableaux(['A',3], shape = [1,1])(rows=[[1],[3]])
sage: K.retract(b)
[[1], [3]]
sage: type(K.retract(b))
<class 'sage.combinat.crystals.kirillov_reshetikhin.KR_type_A_with_category.element_class'>
```

The 0-arrows are obtained using the analogue of σ, called the promotion operator pr, on the level of crystals via:

$$f_0 = \mathrm{pr}^{-1} \circ f_1 \circ \mathrm{pr}$$
$$e_0 = \mathrm{pr}^{-1} \circ e_1 \circ \mathrm{pr}$$

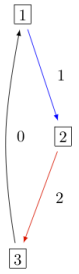
In Sage this can be achieved as follows:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: b = K.module_generator(); b
[[1], [2]]
sage: b.f(0)
sage: b.e(0)
[[2], [4]]

sage: K.promotion()(b.lift())
[[2], [3]]
sage: K.promotion()(b.lift()).e(1)
[[1], [3]]
sage: K.promotion_inverse()(K.promotion()(b.lift()).e(1))
[[2], [4]]
```

KR crystals are level 0 crystals, meaning that the weight of all elements in these crystals is zero:

```
sage: K = crystals.KirillovReshetikhin(['A',3,1],2,1)
sage: b = K.module_generator(); b.weight()
-Lambda[0] + Lambda[2]
sage: b.weight().level()
0
```

The KR crystal $B^{1,1}$ of type $A_2^{(1)}$ looks as follows:



In Sage this can be obtained via:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: G = K.digraph()
sage: view(G, tightpage=True) # optional - dot2tex graphviz
```

Types $D_n^{(1)}$, $B_n^{(1)}$, $A_{2n-1}^{(2)}$

The Dynkin diagrams for types $D_n^{(1)}$, $B_n^{(1)}$, $A_{2n-1}^{(2)}$ are invariant under interchanging nodes 0 and 1:

```
sage: n = 5
sage: C = CartanType(['D',n,1]); C.dynkin_diagram()
  0 O   O 5
    |   |
    |   |
O---O---O---O
1   2   3   4
D5~
sage: C = CartanType(['B',n,1]); C.dynkin_diagram()
    O 0
    |
    |
O---O---O---O=>=O
1   2   3   4   5
B5~
sage: C = CartanType(['A',2*n-1,2]); C.dynkin_diagram()
    O 0
    |
    |
O---O---O---O=<=O
1   2   3   4   5
B5~*
```

The underlying classical algebras obtained when removing node 0 are type $\mathfrak{g}_0 = D_n, B_n, C_n$, respectively. The classical decomposition into a $\mathfrak{g}_0$ crystal is a direct sum:

$$B^{r,s} \cong \bigoplus_{\lambda} B(\lambda) \quad \text{as a } \{1,2,\ldots,n\}\text{-crystal}$$

where λ is obtained from $s\omega_r$ (or equivalently a rectangular partition of shape (s^r)) by removing vertical dominoes. This in fact only holds in the ranges $1 \leq r \leq n-2$ for type $D_n^{(1)}$, and $1 \leq r \leq n$ for types $B_n^{(1)}$ and $A_{2n-1}^{(2)}$:

```
sage: K = crystals.KirillovReshetikhin(['D',6,1],4,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 6] and shape(s)
 [[], [1, 1], [1, 1, 1, 1], [2, 2], [2, 2, 1, 1], [2, 2, 2, 2]]
```

For type $B_n^{(1)}$ and $r = n$, one needs to be aware that ω_n is a spin weight and hence corresponds in the partition language to a column of height n and width $1/2$:

```
sage: K = crystals.KirillovReshetikhin(['B',3,1],3,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[1/2, 1/2, 1/2]]
```

As for type $A_n^{(1)}$, the Dynkin automorphism induces a promotion-type operator σ on the level of crystals. In this case in can however happen that the automorphism changes between classical components:

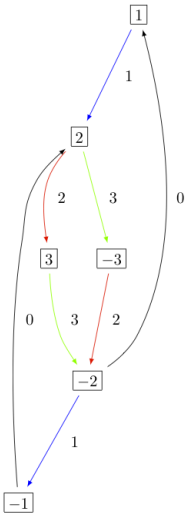
```
sage: K = crystals.KirillovReshetikhin(['D',4,1],2,1)
sage: b = K.module_generator(); b
[[1], [2]]
sage: K.automorphism(b)
[[2], [-1]]
sage: b = K(rows=[[2],[-2]])
```

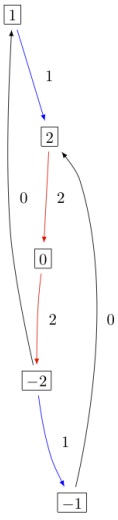
```
sage: K.automorphism(b)
[]
```

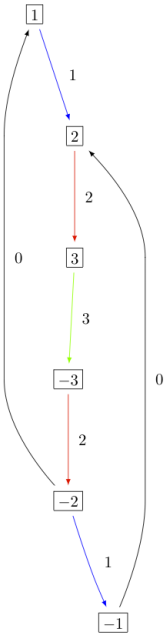
This operator σ is used to define the affine crystal operators:

$$f_0 = \sigma \circ f_1 \circ \sigma$$
$$e_0 = \sigma \circ e_1 \circ \sigma$$

The KR crystals $B^{1,1}$ of types $D_3^{(1)}$, $B_2^{(1)}$, and $A_5^{(2)}$ are, respectively:







Type $C_n^{(1)}$

The Dynkin diagram of type $C_n^{(1)}$ has a symmetry $\sigma(i) = n - i$:

```
sage: C = CartanType(['C',4,1]); C.dynkin_diagram()
O=>=O---O---O=<=O
0   1   2   3   4
C4~
```

The classical subalgebra when removing the 0 node is of type C_n.

However, in this case the crystal $B^{r,s}$ is not constructed using σ, but rather using a virtual crystal construction. $B^{r,s}$ of type $C_n^{(1)}$ is realized inside $\hat{V}^{r,s}$ of type $A_{2n+1}^{(2)}$ using:

$$e_0 = \hat{e}_0\hat{e}_1 \quad \text{and} \quad e_i = \hat{e}_{i+1} \quad \text{for} \quad 1 \leq i \leq n$$
$$f_0 = \hat{f}_0\hat{f}_1 \quad \text{and} \quad f_i = \hat{f}_{i+1} \quad \text{for} \quad 1 \leq i \leq n$$

where $\hat{e}_i$ and $\hat{f}_i$ are the crystal operator in the ambient crystal $\hat{V}^{r,s}$:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],1,2); K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['B', 4, 1]^* with (r,s)=(1,2)
```

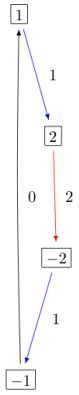
The classical decomposition for $1 \leq r < n$ is given by:

$$B^{r,s} \cong \bigoplus_{\lambda} B(\lambda) \quad \text{as a } \{1, 2, \ldots, n\}\text{-crystal}$$

where λ is obtained from $s\omega_r$ (or equivalently a rectangular partition of shape (s^r)) by removing horizontal dominoes:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],2,4)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 3] and shape(s) [[], [2], [4], [2, 2], [4, 2], [4, 4]]
```

The KR crystal $B^{1,1}$ of type $C_2^{(1)}$ looks as follows:



Types $D_{n+1}^{(2)}$, $A_{2n}^{(2)}$

The Dynkin diagrams of types $D_{n+1}^{(2)}$ and $A_{2n}^{(2)}$ look as follows:

```
sage: C = CartanType(['D',5,2]); C.dynkin_diagram()
O=<=O---O---O=>=O
0   1   2   3   4
C4~*

sage: C = CartanType(['A',8,2]); C.dynkin_diagram()
O=<=O---O---O=<=O
0   1   2   3   4
BC4~
```

The classical subdiagram is of type B_n for type $D_{n+1}^{(2)}$ and of type C_n for type $A_{2n}^{(2)}$. The classical decomposition for these KR crystals for $1 \leq r < n$ for type $D_{n+1}^{(2)}$ and $1 \leq r \leq n$ for type $A_{2n}^{(2)}$ is given by:

$$B^{r,s} \cong \bigoplus_{\lambda} B(\lambda) \quad \text{as a } \{1, 2, \ldots, n\}\text{-crystal}$$

where λ is obtained from $s\omega_r$ (or equivalently a rectangular partition of shape (s^r)) by removing single boxes:

```
sage: K = crystals.KirillovReshetikhin(['D',5,2],2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 4] and shape(s) [[], [1], [2], [1, 1], [2, 1], [2, 2]]

sage: K = crystals.KirillovReshetikhin(['A',8,2],2,2)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 4] and shape(s) [[], [1], [2], [1, 1], [2, 1], [2, 2]]
```

The KR crystals are constructed using an injective map into a KR crystal of type $C_n^{(1)}$

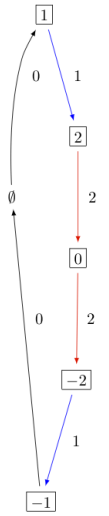
$$S : B^{r,s} \to B^{r,2s}_{C_n^{(1)}} \quad \text{such that } S(e_i b) = e_i^{m_i} S(b) \text{ and } S(f_i b) = f_i^{m_i} S(b)$$

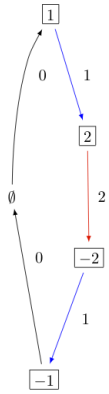
where

$$(m_0, \ldots, m_n) = (1, 2, \ldots, 2, 1) \text{ for type } D_{n+1}^{(2)} \quad \text{and} \quad (1, 2, \ldots, 2, 2) \text{ for type } A_{2n}^{(2)}.$$

```
sage: K = crystals.KirillovReshetikhin(['D',5,2],1,2); K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['C', 4, 1] with (r,s)=(1,4)
sage: K = crystals.KirillovReshetikhin(['A',8,2],1,2); K.ambient_crystal()
Kirillov-Reshetikhin crystal of type ['C', 4, 1] with (r,s)=(1,4)
```

The KR crystals $B^{1,1}$ of type $D_3^{(2)}$ and $A_4^{(2)}$ look as follows:





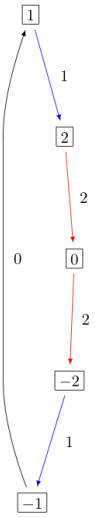
As you can see from the Dynkin diagram for type $A_{2n}^{(2)}$, mapping the nodes $i \mapsto n - i$ yields the same diagram, but with relabelled nodes. In this case the classical subdiagram is of type B_n instead of C_n. One can also construct the KR crystal $B^{r,s}$ of type $A_{2n}^{(2)}$ based on this classical decomposition. In this case the classical decomposition is the sum over all weights obtained from $s\omega_r$ by removing horizontal dominoes:

```
sage: C = CartanType(['A',6,2]).dual()
sage: Kdual = crystals.KirillovReshetikhin(C,2,2)
sage: Kdual.classical_decomposition()
The crystal of tableaux of type ['B', 3] and shape(s) [[], [2], [2, 2]]
```

Looking at the picture, one can see that this implementation is isomorphic to the other implementation based on the C_n decomposition up to a relabeling of the arrows:

```
sage: C = CartanType(['A',4,2])
sage: K = crystals.KirillovReshetikhin(C,1,1)
sage: Kdual = crystals.KirillovReshetikhin(C.dual(),1,1)
sage: G = K.digraph()
sage: Gdual = Kdual.digraph()
```

```
sage: f = { 1:1, 0:2, 2:0 }
sage: for u,v,label in Gdual.edges():
....:     Gdual.set_edge_label(u,v,f[label])
sage: G.is_isomorphic(Gdual, edge_labels = True, certify = True)
(True, {[[-2]]: [[1]], [[-1]]: [[2]], [[1]]: [[-2]], []: [[0]], [[2]]: [[-1]]})
```



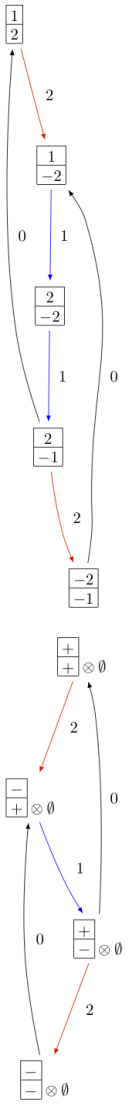
Exceptional nodes

The KR crystals $B^{n,s}$ for types $C_n^{(1)}$ and $D_{n+1}^{(2)}$ were excluded from the above discussion. They are associated to the exceptional node $r = n$ and in this case the classical decomposition is irreducible:

$$B^{n,s} \cong B(s\omega_n).$$

In Sage:

```
sage: K = crystals.KirillovReshetikhin(['C',2,1],2,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['C', 2] and shape(s) [[1, 1]]

sage: K = crystals.KirillovReshetikhin(['D',3,2],2,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['B', 2] and shape(s) [[1/2, 1/2]]
```



The KR crystals $B^{n,s}$ and $B^{n-1,s}$ of type $D_n^{(1)}$ are also special. They decompose as:

$$B^{n,s} \cong B(s\omega_n) \quad \text{and} \quad B^{n-1,s} \cong B(s\omega_{n-1}).$$

```
sage: K = crystals.KirillovReshetikhin(['D',4,1],4,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, 1/2]]
sage: K = crystals.KirillovReshetikhin(['D',4,1],3,1)
sage: K.classical_decomposition()
The crystal of tableaux of type ['D', 4] and shape(s) [[1/2, 1/2, 1/2, -1/2]]
```

Type $E_6^{(1)}$

In [JonesEtAl2010] the KR crystals $B^{r,s}$ for $r = 1, 2, 6$ in type $E_6^{(1)}$ were constructed exploiting again a Dynkin diagram automorphism, namely the automorphism σ of order 3 which maps $0 \mapsto 1 \mapsto 6 \mapsto 0$:

```
sage: C = CartanType(['E',6,1]); C.dynkin_diagram()
        O 0
        |
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
```

The crystals $B^{1,s}$ and $B^{6,s}$ are irreducible as classical crystals:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.classical_decomposition()
Direct sum of the crystals Family (Finite dimensional highest weight crystal of type ['E', 6] and hig
sage: K = crystals.KirillovReshetikhin(['E',6,1],6,1)
sage: K.classical_decomposition()
Direct sum of the crystals Family (Finite dimensional highest weight crystal of type ['E', 6] and hig
```

whereas for the adjoint node $r = 2$ we have the decomposition

$$B^{2,s} \cong \bigoplus_{k=0}^{s} B(k\omega_2)$$

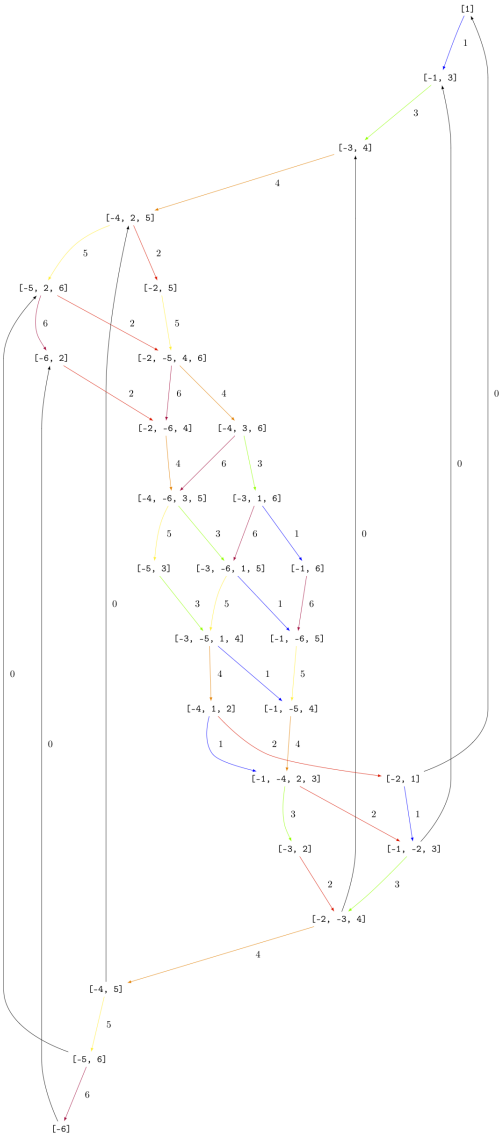
```
sage: K = crystals.KirillovReshetikhin(['E',6,1],2,1)
sage: K.classical_decomposition()
Direct sum of the crystals Family (Finite dimensional highest weight crystal of type ['E', 6] and hig
Finite dimensional highest weight crystal of type ['E', 6] and highest weight Lambda[2])
```

The promotion operator on the crystal corresponding to σ can be calculated explicitly:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: promotion = K.promotion()
sage: u = K.module_generator(); u
[(1,)]
sage: promotion(u.lift())
[(-1, 6)]
```

The crystal $B^{1,1}$ is already of dimension 27. The elements b of this crystal are labelled by tuples which specify their nonzero $\phi_i(b)$ and $\epsilon_i(b)$. For example, $[-6, 2]$ indicates that $\phi_2([-6, 2]) = \epsilon_6([-6, 2]) = 1$ and all others are equal to zero:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1],1,1)
sage: K.cardinality()
27
```



Single column KR crystals

A single column KR crystal is $B^{r,1}$ for any $r \in I_0$.

In [LNSSS14I] and [LNSSS14II], it was shown that single column KR crystals can be constructed by projecting level 0 crystals of LS paths onto the classical weight lattice. We first verify that we do get an isomorphic crystal for $B^{1,1}$ in type $E_6^{(1)}$:

```
sage: K = crystals.KirillovReshetikhin(['E',6,1], 1,1)
sage: K2 = crystals.kirillov_reshetikhin.LSPaths(['E',6,1], 1,1)
sage: K.digraph().is_isomorphic(K2.digraph(), edge_labels=True)
True
```

Here is an example in $E_8^{(1)}$ and we calculate its classical decomposition:

```
sage: K = crystals.kirillov_reshetikhin.LSPaths(['E',8,1], 8,1)
sage: K.cardinality()
249
```

```
sage: L = [x for x in K if x.is_highest_weight([1,2,3,4,5,6,7,8])]
sage: map(lambda x: x.weight(), L)
[-2*Lambda[0] + Lambda[8], 0]
```

Applications

An important notion for finite-dimensional affine crystals is perfectness. The crucial property is that a crystal B is perfect of level ℓ if there is a bijection between level ℓ dominant weights and elements in

$$B_{\min} = \{b \in B \mid \mathrm{lev}(\varphi(b)) = \ell\} .$$

For a precise definition of perfect crystals see [HongKang2002] . In [FourierEtAl2010] it was proven that for the nonexceptional types $B^{r,s}$ is perfect as long as s/c_r is an integer. Here $c_r = 1$ except $c_r = 2$ for $1 \leq r < n$ in type $C_n^{(1)}$ and $r = n$ in type $B_n^{(1)}$.

Here we verify this using Sage for $B^{1,1}$ of type $C_3^{(1)}$:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],1,1)
sage: Lambda = K.weight_lattice_realization().fundamental_weights(); Lambda
Finite family {0: Lambda[0], 1: Lambda[1], 2: Lambda[2], 3: Lambda[3]}
sage: [w.level() for w in Lambda]
[1, 1, 1, 1]
sage: Bmin = [b for b in K if b.Phi().level() == 1 ]; Bmin
[[[1]], [[2]], [[3]], [[-3]], [[-2]], [[-1]]]
sage: [b.Phi() for b in Bmin]
[Lambda[1], Lambda[2], Lambda[3], Lambda[2], Lambda[1], Lambda[0]]
```

As you can see, both $b = 1$ and $b = -2$ satisfy $\varphi(b) = \Lambda_1$. Hence there is no bijection between the minimal elements in $B_{\min}$ and level 1 weights. Therefore, $B^{1,1}$ of type $C_3^{(1)}$ is not perfect. However, $B^{1,2}$ of type $C_n^{(1)}$ is a perfect crystal:

```
sage: K = crystals.KirillovReshetikhin(['C',3,1],1,2)
sage: Lambda = K.weight_lattice_realization().fundamental_weights()
sage: Bmin = [b for b in K if b.Phi().level() == 1 ]
sage: [b.Phi() for b in Bmin]
[Lambda[0], Lambda[3], Lambda[2], Lambda[1]]
```

Perfect crystals can be used to construct infinite-dimensional highest weight crystals and Demazure crystals using the Kyoto path model [KKMMNN1992]. We construct Example 10.6.5 in [HongKang2002]:

```
sage: K = crystals.KirillovReshetikhin(['A',1,1], 1,1)
sage: La = RootSystem(['A',1,1]).weight_lattice().fundamental_weights()
sage: B = crystals.KyotoPathModel(K, La[0])
sage: B.highest_weight_vector()
[[[2]]]

sage: K = crystals.KirillovReshetikhin(['A',2,1], 1,1)
sage: La = RootSystem(['A',2,1]).weight_lattice().fundamental_weights()
sage: B = crystals.KyotoPathModel(K, La[0])
sage: B.highest_weight_vector()
[[[3]]]

sage: K = crystals.KirillovReshetikhin(['C',2,1], 2,1)
sage: La = RootSystem(['C',2,1]).weight_lattice().fundamental_weights()
sage: B = crystals.KyotoPathModel(K, La[1])
```

```
sage: B.highest_weight_vector()
[[[2], [-2]]]
```

Energy function and one-dimensional configuration sum

For tensor products of Kirillov-Reshehtikhin crystals, there also exists the important notion of the energy function. It can be defined as the sum of certain local energy functions and the R-matrix. In Theorem 7.5 in [SchillingTingley2011] it was shown that for perfect crystals of the same level the energy $D(b)$ is the same as the affine grading (up to a normalization). The affine grading is defined as the minimal number of applications of e_0 to b to reach a ground state path. Computationally, this algorithm is a lot more efficient than the computation involving the R-matrix and has been implemented in Sage:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K,K)
sage: hw = [b for b in T if all(b.epsilon(i)==0 for i in [1,2])]
sage: for b in hw:
....:     print b, b.energy_function()
[[[1]], [[1]], [[1]]] 0
[[[1]], [[2]], [[1]]] 2
[[[2]], [[1]], [[1]]] 1
[[[3]], [[2]], [[1]]] 3
```

The affine grading can be computed even for nonperfect crystals:

```
sage: K = crystals.KirillovReshetikhin(['C',4,1],1,2)
sage: K1 = crystals.KirillovReshetikhin(['C',4,1],1,1)
sage: T = crystals.TensorProduct(K,K1)
sage: hw = [b for b in T if all(b.epsilon(i)==0 for i in [1,2,3,4])]
sage: for b in hw:
....:     print b, b.affine_grading()
....:
[[], [[1]]] 1
[[[1, 1]], [[1]]] 2
[[[1, 2]], [[1]]] 1
[[[1, -1]], [[1]]] 0
```

The one-dimensional configuration sum of a crystal B is the graded sum by energy of the weight of all elements $b \in B$:

$$X(B) = \sum_{b \in B} x^{\mathrm{weight}(b)} q^{D(b)}$$

Here is an example of how you can compute the one-dimensional configuration sum in Sage:

```
sage: K = crystals.KirillovReshetikhin(['A',2,1],1,1)
sage: T = crystals.TensorProduct(K,K)
sage: T.one_dimensional_configuration_sum()
B[-2*Lambda[1] + 2*Lambda[2]] + (q+1)*B[-Lambda[1]]
 + (q+1)*B[Lambda[1] - Lambda[2]] + B[2*Lambda[1]]
 + B[-2*Lambda[2]] + (q+1)*B[Lambda[2]]
```

Affine Highest Weight Crystals

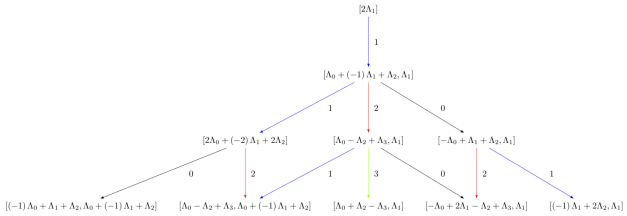
Affine highest weight crystals are infinite-dimensional. Hence, to work with them in Sage, we need some further tools.

Littelmann path model

The Littelmann path model for highest weight crystals is implemented in Sage. It models finite highest crystals as well as affine highest weight crystals which are infinite dimensional. The elements of the crystal are piecewise linear maps in the weight space. For more information on the Littelmann path model, see [L1995].

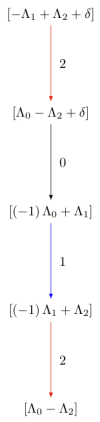
Since the affine highest weight crystals are infinite, it is not possible to list all elements or draw the entire crystal graph. However, if the user is only interested in the crystal up to a certain distance or depth from the highest weight element, then one can work with the corresponding subcrystal. To view the corresponding upper part of the crystal, one can build the associated digraph:

```
sage: R = RootSystem(['C',3,1])
sage: La = R.weight_space().basis()
sage: LS = crystals.LSPaths(2*La[1]); LS
The crystal of LS paths of type ['C', 3, 1] and weight 2*Lambda[1]
sage: C = LS.subcrystal(max_depth=3)
sage: sorted(C, key=str)
[(-Lambda[0] + 2*Lambda[1] - Lambda[2] + Lambda[3], Lambda[1]),
 (-Lambda[0] + Lambda[1] + Lambda[2], Lambda[0] - Lambda[1] + Lambda[2]),
 (-Lambda[0] + Lambda[1] + Lambda[2], Lambda[1]),
 (-Lambda[1] + 2*Lambda[2], Lambda[1]),
 (2*Lambda[0] - 2*Lambda[1] + 2*Lambda[2],),
 (2*Lambda[1],),
 (Lambda[0] + Lambda[2] - Lambda[3], Lambda[1]),
 (Lambda[0] - Lambda[1] + Lambda[2], Lambda[1]),
 (Lambda[0] - Lambda[2] + Lambda[3], Lambda[0] - Lambda[1] + Lambda[2]),
 (Lambda[0] - Lambda[2] + Lambda[3], Lambda[1])]

sage: G = LS.digraph(subset = C)
sage: view(G, pdflatex=True, tightpage=True)  #optional - dot2tex graphviz
```



The Littelmann path model also lends itself as a model for level zero crystals which are bi-infinite. To cut out a slice of these crystals, one can use the direction option in subcrystal:

```
sage: R = RootSystem(['A',2,1])
sage: La = R.weight_space(extended = True).basis()
sage: LS = crystals.LSPaths(La[1]-La[0]); LS
The crystal of LS paths of type ['A', 2, 1] and weight -Lambda[0] + Lambda[1]
sage: C = LS.subcrystal(max_depth=2, direction = 'both')
sage: G = LS.digraph(subset = C)
sage: G.set_latex_options(edge_options = lambda (u,v,label): ({}))
sage: view(G, pdflatex=True, tightpage=True)  #optional - dot2tex graphviz
```



Modified Nakajima monomials

Modified Nakajima monomials have also been implemented in Sage and models highest weight crystals in all symmetrizable types. The elements are given in terms of commuting variables $Y_i(n)$ where $i \in I$ and $n \in \mathbf{Z}_{\geq 0}$. For more information on the modified Nakajima monomials, see [KKS2007].

We give an example in affine type and verify that up to depth 3, it agrees with the Littelmann path model:

```
sage: La = RootSystem(['C',3,1]).weight_space().fundamental_weights()
sage: LS = crystals.LSPaths(2*La[1]+La[2])
sage: SL = LS.subcrystal(max_depth=3)
sage: GL = LS.digraph(subset=SL)

sage: La = RootSystem(['C',3,1]).weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(['C',3,1], 2*La[1]+La[2])
sage: SM = M.subcrystal(max_depth=3)
sage: GM = M.digraph(subset=SM)
sage: GL.is_isomorphic(GM, edge_labels=True)
True
```

Now we do an example of a simply-laced (so symmetrizable) hyperbolic type $H_1^{(4)}$, which comes from the complete graph on 4 vertices:

```
sage: CM = CartanMatrix([[2, -1, -1,-1],[-1,2,-1,-1],[-1,-1,2,-1],[-1,-1,-1,2]]); CM
[ 2 -1 -1 -1]
[-1  2 -1 -1]
[-1 -1  2 -1]
[-1 -1 -1  2]
sage: La = RootSystem(CM).weight_lattice().fundamental_weights()
sage: M = crystals.NakajimaMonomials(CM, La[0])
sage: SM = M.subcrystal(max_depth=4)
sage: GM = M.digraph(subset=SM) # long time
```


Iwahori Hecke Algebras

The `Iwahori Hecke algebra` is defined in [Iwahori1964]. In that original paper, the algebra occurs as the convolution ring of functions on a p-adic group that are compactly supported and invariant both left and right by the Iwahori subgroup. However Iwahori determined its structure in terms of generators and relations, and it turns out to be a deformation of the group algebra of the affine Weyl group.

Once the presentation is found, the Iwahori Hecke algebra can be defined for any Coxeter group. It depends on a parameter q which in Iwahori's paper is the cardinality of the residue field. But it could just as easily be an indeterminate.

Then the Iwahori Hecke algebra has the following description. Let W be a Coxeter group, with generators (simple reflections) $s_1, \ldots, s_n$. They satisfy the relations $s_i^2 = 1$ and the braid relations

$$s_i s_j s_i s_j \cdots = s_j s_i s_j s_i \cdots$$

where the number of terms on each side is the order of $s_i s_j$.

The Iwahori Hecke algebra has a basis $T_1, \ldots, T_n$ subject to relations that resemble those of the s_i. They satisfy the braid relations and the quadratic relation

$$(T_i - q)(T_i + 1) = 0.$$

This can be modified by letting q_1 and q_2 be two indeterminates and letting

$$(T_i - q_1)(T_i - q_2) = 0.$$

In this generality, Iwahori Hecke algebras have significance far beyond their origin in the representation theory of p-adic groups. For example, they appear in the geometry of Schubert varieties, where they are used in the definition of the Kazhdan-Lusztig polynomials. They appear in connection with quantum groups, and in Jones's original paper on the Jones polynomial.

Here is how to create an Iwahori Hecke algebra (in the T basis):

```
sage: R.<q> = PolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra("B3",q)
sage: T = H.T(); T
Iwahori-Hecke algebra of type B3 in q,-1 over Univariate Polynomial Ring
 in q over Integer Ring in the T-basis
sage: T1,T2,T3 = T.algebra_generators()
sage: T1*T1
(q-1)*T[1] + q
```

If the Cartan type is affine, the generators will be numbered starting with `T0` instead of `T1`.

You may coerce a Weyl group element into the Iwahori Hecke algebra:

```
sage: W = WeylGroup("G2",prefix="s")
sage: [s1,s2] = W.simple_reflections()
sage: P.<q> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra("B3",q)
sage: T = H.T()
sage: T(s1*s2)
T[1,2]
```

Kazhdan-Lusztig Polynomials

Sage can compute ordinary Kazhdan-Lusztig polynomials for Weyl groups or affine Weyl groups (and potentially other Coxeter groups).

You must create a Weyl group `W` and a ring containing an indeterminate `q`. The ring may be a univariate polynomial ring or a univariate Laurent polynomial ring. Then you may calculate Kazhdan-Lusztig polynomials as follows:

```
sage: W = WeylGroup("A3", prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: P.<q> = LaurentPolynomialRing(QQ)
sage: KL = KazhdanLusztigPolynomial(W,q)
```

```
sage: KL.R(s2, s2*s1*s3*s2)
-1 + 3*q - 3*q^2 + q^3
sage: KL.P(s2, s2*s1*s3*s2)
1 + q
```

Thus we have the Kazhdan-Lusztig R and P polynomials.

Known algorithms for computing Kazhdan-Lusztig polynomials are highly recursive, and caching of intermediate results is necessary for the programs not to be prohibitively slow. Therefore intermediate results are cached. This has the effect that as you run the program for any given `KazhdanLusztigPolynomial` class, the calculations will be slow at first but progressively faster as more polynomials are computed.

You may see the results of the intermediate calculations by creating the class with the option `trace="true"`.

Since the parent of `q` must be a univariate ring, if you want to work with other indeterminates, *first* create a univariate polynomial or Laurent polynomial ring, and the Kazhdan-Lusztig class. *Then* create a ring containing `q` and the other variables:

```
sage: W = WeylGroup("B3", prefix="s")
sage: [s1,s2,s3] = W.simple_reflections()
sage: P.<q> = PolynomialRing(QQ)
sage: KL = KazhdanLusztigPolynomial(W,q)
sage: P1.<x,y> = PolynomialRing(P)
sage: x*KL.P(s1*s3,s1*s3*s2*s1*s3)
(q + 1)*x
```

Bibliography

6.1.6 Linear Programming (Mixed Integer)

This document explains the use of linear programming (LP) – and of mixed integer linear programming (MILP) – in Sage by illustrating it with several problems it can solve. Most of the examples given are motivated by graph-theoretic concerns, and should be understandable without any specific knowledge of this field. As a tool in Combinatorics, using linear programming amounts to understanding how to reformulate an optimization (or existence) problem through linear constraints.

This is a translation of a chapter from the book Calcul mathematique avec Sage.

Definition

Here we present the usual definition of what a linear program is: it is defined by a matrix $A : \mathbb{R}^m \mapsto \mathbb{R}^n$, along with two vectors $b, c \in \mathbb{R}^n$. Solving a linear program is searching for a vector x maximizing an *objective* function and satisfying a set of constraints, i.e.

$$c^t x = \max_{x' \text{ such that } Ax' \leq b} c^t x'$$

where the ordering $u \leq u'$ between two vectors means that the entries of u' are pairwise greater than the entries of u. We also write:

$$\text{Max: } c^t x$$
$$\text{Such that: } Ax \leq b$$

Equivalently, we can also say that solving a linear program amounts to maximizing a linear function defined over a polytope (preimage or $A^{-1}(\leq b)$). These definitions, however, do not tell us how to use linear programming in combinatorics. In the following, we will show how to solve optimization problems like the Knapsack problem, the Maximum Matching problem, and a Flow problem.

Mixed integer linear programming

There is a bad news coming along with this definition of linear programming: an LP can be solved in polynomial time. This is indeed a bad news, because this would mean that unless we define LP of exponential size, we can not expect LP to solve NP-complete problems, which would be a disappointment. On a brighter side, it becomes NP-complete to solve a linear program if we are allowed to specify constraints of a different kind: requiring that some variables be integers instead of real values. Such an LP is actually called a "mixed integer linear program" (some variables can be integers, some other reals). Hence, we can expect to find in the MILP framework a *wide* range of expressivity.

Practical

The `MILP` class

The `MILP` class in Sage represents a MILP! It is also used to solve regular LP. It has a very small number of methods, meant to define our set of constraints and variables, then to read the solution found by the solvers once computed. It is also possible to export a MILP defined with Sage to a `.lp` or `.mps` file, understood by most solvers.

Let us ask Sage to solve the following LP:

$$\begin{aligned} \text{Max: } & x + y - 3z \\ \text{Such that: } & x + 2y \leq 4 \\ & 5z - y \leq 8 \\ & x, y, z \geq 0 \end{aligned}$$

To achieve it, we need to define a corresponding `MILP` object, along with 3 variables `x`, `y` and `z`:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable(real=True, nonnegative=True)
sage: x, y, z = v['x'], v['y'], v['z']
```

Next, we set the objective function

```
sage: p.set_objective(x + y + 3*z)
```

And finally we set the constraints

```
sage: p.add_constraint(x + 2*y <= 4)
sage: p.add_constraint(5*z - y <= 8)
```

The `solve` method returns by default the optimal value reached by the objective function

```
sage: round(p.solve(), 2)
8.8
```

We can read the optimal assignation found by the solver for x, y and z through the `get_values` method

```
sage: round(p.get_values(x), 2)
4.0
sage: round(p.get_values(y), 2)
0.0
sage: round(p.get_values(z), 2)
1.6
```

Variables

In the previous example, we obtained variables through `v['x'], v['y']` and `v['z']`. This being said, larger LP/MILP will require us to associate a LP variable to many Sage objects, which can be integers, strings, or even the vertices and edges of a graph. For example:

```
sage: x = p.new_variable(real=True, nonnegative=True)
```

With this new object `x` we can now write constraints using `x[1],...,x[15]`.

```
sage: p.add_constraint(x[1] + x[12] - x[14] >= 8)
```

Notice that we did not need to define the "length" of `x`. Actually, `x` would accept any immutable object as a key, as a dictionary would. We can now write

```
sage: p.add_constraint(x["I am a valid key"] +
....:                  x[("a",pi)] <= 3)
```

And because any immutable object can be used as a key, doubly indexed variables $x^{1,1}, ..., x^{1,15}, x^{2,1}, ..., x^{15,15}$ can be referenced by `x[1,1],...,x[1,15],x[2,1],...,x[15,15]`

```
sage: p.add_constraint(x[3,2] + x[5] == 6)
```

Typed variables and bounds **Types :** If you want a variable to assume only integer or binary values, use the `integer=True` or `binary=True` arguments of the `new_variable` method. Alternatively, call the `set_integer` and `set_binary` methods.

Bounds : If you want your variables to only take nonnegative values, you can say so when calling `new_variable` with the argument `nonnegative=True`. If you want to set a different upper/lower bound on a variable, add a constraint or use the use the `set_min`, `set_max` methods.

Basic linear programs

Knapsack

The *Knapsack* problem is the following: given a collection of items having both a weight and a *usefulness*, we would like to fill a bag whose capacity is constrained while maximizing the usefulness of the items contained in the bag (we will consider the sum of the items' usefulness). For the purpose of this tutorial, we set the restriction that the bag can only carry a certain total weight.

To achieve this, we have to associate to each object o of our collection C a binary variable `taken[o]`, set to 1 when

the object is in the bag, and to 0 otherwise. We are trying to solve the following MILP

$$\text{Max: } \sum_{o \in L} \text{usefulness}_o \times \text{taken}_o$$
$$\text{Such that: } \sum_{o \in L} \text{weight}_o \times \text{taken}_o \leq C$$

Using Sage, we will give to our items a random weight:

```
sage: C = 1

sage: L = ["pan", "book", "knife", "gourd", "flashlight"]

sage: L.extend(["random_stuff_" + str(i) for i in range(20)])

sage: weight = {}
sage: usefulness = {}

sage: set_random_seed(685474)
sage: for o in L:
...       weight[o] = random()
...       usefulness[o] = random()
```

We can now define the MILP itself

```
sage: p = MixedIntegerLinearProgram()
sage: taken = p.new_variable(binary=True)

sage: p.add_constraint(sum(weight[o] * taken[o] for o in L) <= C)

sage: p.set_objective(sum(usefulness[o] * taken[o] for o in L))

sage: p.solve() # abs tol 1e-6
3.1502766806530307
sage: taken = p.get_values(taken)
```

The solution found is (of course) admissible

```
sage: sum(weight[o] * taken[o] for o in L) # abs tol 1e-6
0.6964959796619171
```

Should we take a flashlight?

```
sage: taken["flashlight"]
1.0
```

Wise advice. Based on purely random considerations.

Matching

Given a graph G, a matching is a set of pairwise disjoint edges. The empty set is a trivial matching. So we focus our attention on maximum matchings: we want to find in a graph a matching whose cardinality is maximal. Computing the maximum matching in a graph is a polynomial problem, which is a famous result of Edmonds. Edmonds' algorithm is

based on local improvements and the proof that a given matching is maximum if it cannot be improved. This algorithm is not the hardest to implement among those graph theory can offer, though this problem can be modeled with a very simple MILP.

To do it, we need – as previously – to associate a binary variable to each one of our objects: the edges of our graph (a value of 1 meaning that the corresponding edge is included in the maximum matching). Our constraint on the edges taken being that they are disjoint, it is enough to require that, x and y being two edges and m_x, m_y their associated variables, the inequality $m_x + m_y \leq 1$ is satisfied, as we are sure that the two of them cannot both belong to the matching. Hence, we are able to write the MILP we want. However, the number of inequalities can be easily decreased by noticing that two edges cannot be taken simultaneously inside a matching if and only if they have a common endpoint v. We can then require instead that at most one edge incident to v be taken inside the matching, which is a linear constraint. We will be solving:

$$\text{Max: } \sum_{e \in E(G)} m_e$$

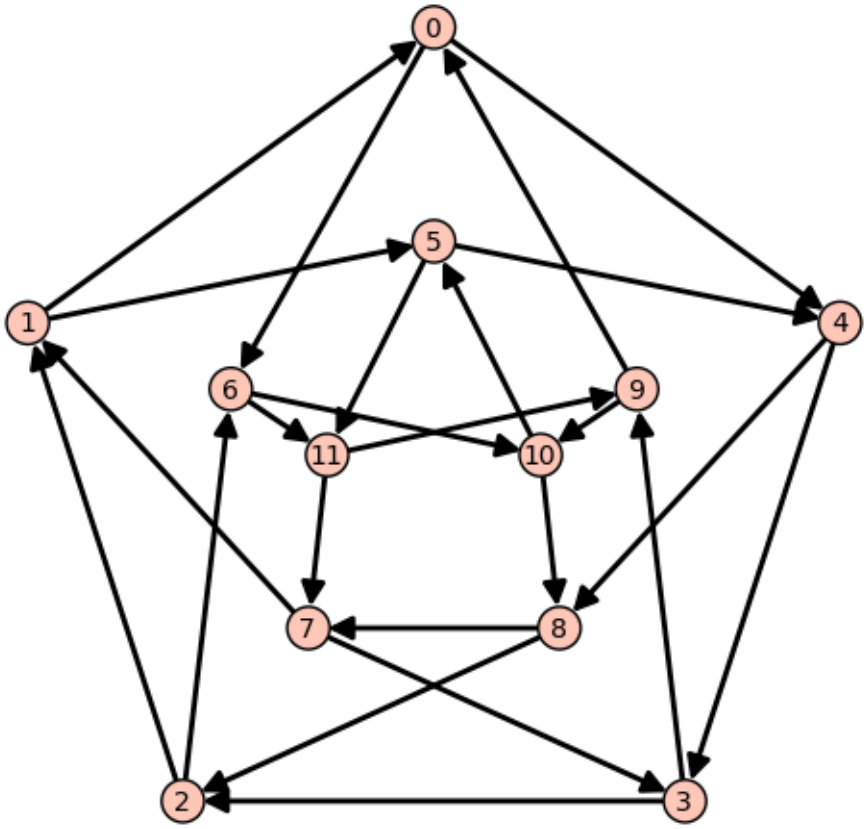
$$\text{Such that: } \forall v, \sum_{\substack{e \in E(G) \\ v \sim e}} m_e \leq 1$$

Let us write the Sage code of this MILP:

```
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram()
sage: matching = p.new_variable(binary=True)

sage: p.set_objective(sum(matching[e] for e in g.edges(labels=False)))

sage: for v in g:
...       p.add_constraint(sum(matching[e]
...           for e in g.edges_incident(v, labels=False)) <= 1)

sage: p.solve()
5.0

sage: matching = p.get_values(matching)
sage: [e for e,b in matching.iteritems() if b == 1]  # not tested
[(0, 1), (6, 9), (2, 7), (3, 4), (5, 8)]
```

Flows

Yet another fundamental algorithm in graph theory: maximum flow! It consists, given a directed graph and two vertices s, t, in sending a maximum *flow* from s to t using the edges of G, each of them having a maximal capacity.



The definition of this problem is almost its LP formulation. We are looking for real values associated to each edge, which would represent the intensity of flow going through them, under two types of constraints:

- The amount of flow arriving on a vertex (different from s or t) is equal to the amount of flow leaving it.
- The amount of flow going through an edge is bounded by the capacity of this edge.

This being said, we have to maximize the amount of flow leaving s: all of it will end up in t, as the other vertices are sending just as much as they receive. We can model the flow problem with the following LP

$$\text{Max: } \sum_{sv \in G} f_{sv}$$

$$\text{Such that: } \forall v \in G, \begin{matrix} v \neq s \\ v \neq t \end{matrix}, \sum_{vu \in G} f_{vu} - \sum_{uv \in G} f_{uv} = 0$$

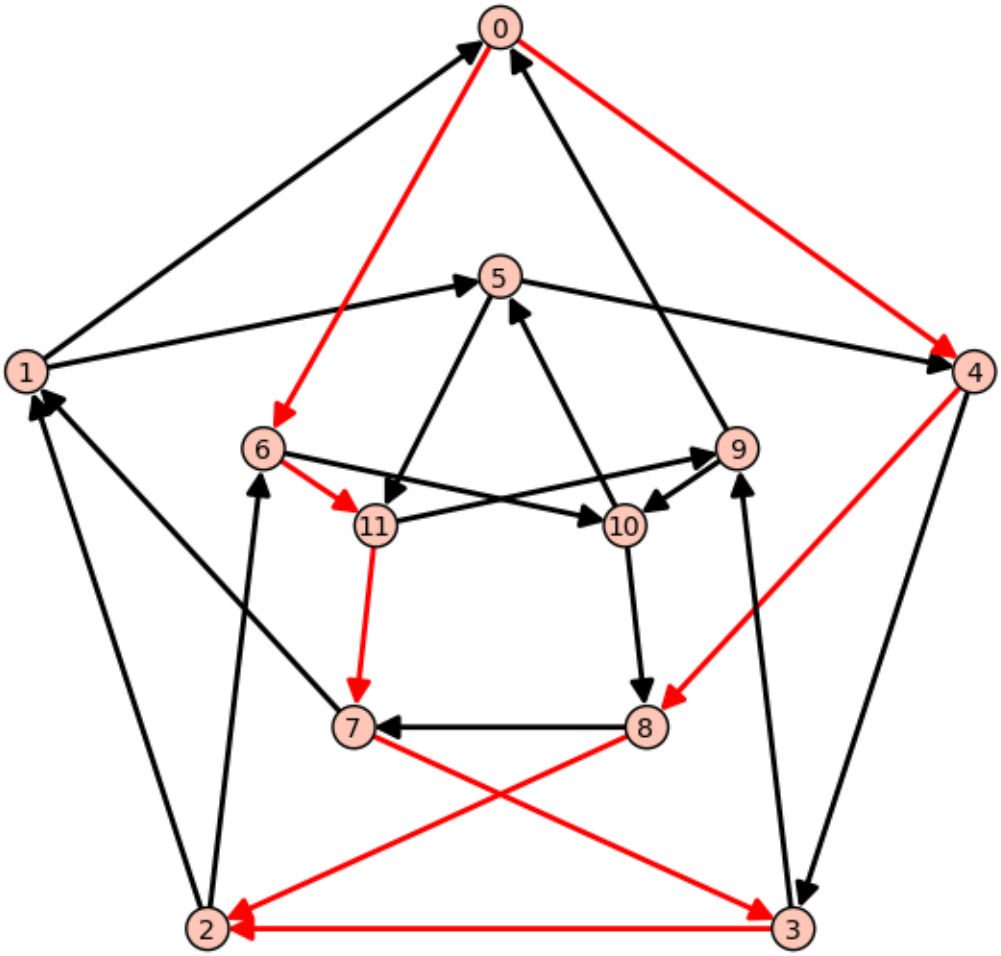
$$\forall uv \in G, f_{uv} \leq 1$$

We will solve the flow problem on an orientation of Chvatal's graph, in which all the edges have a capacity of 1:

```
sage: g = graphs.ChvatalGraph()
sage: g = g.minimum_outdegree_orientation()
```

```
sage: p = MixedIntegerLinearProgram()
sage: f = p.new_variable(real=True, nonnegative=True)
sage: s, t = 0, 2
```

```
sage: for v in g:
...       if v != s and v != t:
...           p.add_constraint(
...               sum(f[(v,u)] for u in g.neighbors_out(v))
...               - sum(f[(u,v)] for u in g.neighbors_in(v)) == 0)

sage: for e in g.edges(labels=False):
...       p.add_constraint(f[e] <= 1)

sage: p.set_objective(sum(f[(s,u)] for u in g.neighbors_out(s)))

sage: p.solve()
2.0
```



Solvers

Sage solves linear programs by calling specific libraries. The following libraries are currently supported:

- CBC: A solver from COIN-OR

Provided under the open source license CPL, but incompatible with GPL. CBC can be installed through the command `install_package("cbc")`.

- CPLEX: A solver from ILOG

  Proprietary, but free for researchers and students.

- CVXOPT: an LP solver from Python Software for Convex Optimization, uses an interior-point method.

  Licensed under the GPL.

- GLPK: A solver from GNU

  Licensed under the GPLv3. This solver is installed by default with Sage.

- GUROBI

  Proprietary, but free for researchers and students.

- PPL: A solver from bugSeng.

  Licensed under the GPLv3. This solver provides exact (arbitrary precision) computation.

Using CPLEX or GUROBI through Sage

ILOG's CPLEX and GUROBI being proprietary softwares, you must be in possession of several files to use it through Sage. In each case, the **expected** (it may change !) filename is joined.

- **A valid license file**
  - CPLEX : a `.ilm` file
  - GUROBI : a `.lic` file
- **A compiled version of the library**
  - CPLEX : `libcplex.a`
  - GUROBI : `libgurobi55.so` (or more recent)
- **The library file**
  - CPLEX : `cplex.h`
  - GUROBI : `gurobi_c.h`

The environment variable defining the licence's path must also be set when running Sage. You can append to your `.bashrc` file one of the following :

- For CPLEX

```
export ILOG_LICENSE_FILE=/path/to/the/license/ilog/ilm/access_1.ilm
```

- For GUROBI

```
export GRB_LICENSE_FILE=/path/to/the/license/gurobi.lic
```

As Sage also needs the files library and header files the easiest way is to create symbolic links to these files in the appropriate directories:

- **For CPLEX:**
  - `libcplex.a` – in `SAGE_ROOT/local/lib/`, type:

```
ln -s /path/to/lib/libcplex.a .
```

  - `cplex.h` – in `SAGE_ROOT/local/include/`, type:

    ```
    ln -s /path/to/include/cplex.h .
    ```

  - `cpxconst.h` (if it exists) – in `SAGE_ROOT/local/include/`, type:

    ```
    ln -s /path/to/include/cpxconst.h .
    ```

- For GUROBI
  - `libgurobi45.so` – in `SAGE_ROOT/local/lib/`, type:

    ```
    ln -s /path/to/lib/libgurobi45.so libgurobi.so
    ```

  - `gurobi_c.h` – in `SAGE_ROOT/local/include/`, type:

    ```
    ln -s /path/to/include/gurobi_c.h .
    ```

It is very important that the names of the symbolic links in Sage's folders ** be precisely as indicated. If the names differ, Sage will not notice that** **the files are present**

Once this is done, Sage is to be asked to notice the changes by calling:

```
sage -b
```

6.1.7 Number Theory and the RSA Public Key Cryptosystem

Author: Minh Van Nguyen <nguyenminh2@gmail.com>

This tutorial uses Sage to study elementary number theory and the RSA public key cryptosystem. A number of Sage commands will be presented that help us to perform basic number theoretic operations such as greatest common divisor and Euler's phi function. We then present the RSA cryptosystem and use Sage's built-in commands to encrypt and decrypt data via the RSA algorithm. Note that this tutorial on RSA is for pedagogy purposes only. For further details on cryptography or the security of various cryptosystems, consult specialized texts such as [MenezesEtAl1996], [Stinson2006], and [TrappeWashington2006].

Elementary number theory

We first review basic concepts from elementary number theory, including the notion of primes, greatest common divisors, congruences and Euler's phi function. The number theoretic concepts and Sage commands introduced will be referred to in later sections when we present the RSA algorithm.

Prime numbers

Public key cryptography uses many fundamental concepts from number theory, such as prime numbers and greatest common divisors. A positive integer $n > 1$ is said to be *prime* if its factors are exclusively 1 and itself. In Sage, we can obtain the first 20 prime numbers using the command `primes_first_n`:

```
sage: primes_first_n(20)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

Greatest common divisors

Let a and b be integers, not both zero. Then the greatest common divisor (GCD) of a and b is the largest positive integer which is a factor of both a and b. We use $\gcd(a, b)$ to denote this largest positive factor. One can extend this definition by setting $\gcd(0, 0) = 0$. Sage uses `gcd(a, b)` to denote the GCD of a and b. The GCD of any two distinct primes is 1, and the GCD of 18 and 27 is 9.

```
sage: gcd(3, 59)
1
sage: gcd(18, 27)
9
```

If $\gcd(a, b) = 1$, we say that a is *coprime* (or relatively prime) to b. In particular, $\gcd(3, 59) = 1$ so 3 is coprime to 59 and vice versa.

Congruences

When one integer is divided by a non-zero integer, we usually get a remainder. For example, upon dividing 23 by 5, we get a remainder of 3; when 8 is divided by 5, the remainder is again 3. The notion of congruence helps us to describe the situation in which two integers have the same remainder upon division by a non-zero integer. Let $a, b, n \in \mathbf{Z}$ such that $n \neq 0$. If a and b have the same remainder upon division by n, then we say that a is *congruent* to b modulo n and denote this relationship by

$$a \equiv b \pmod{n}$$

This definition is equivalent to saying that n divides the difference of a and b, i.e. $n \mid (a - b)$. Thus $23 \equiv 8 \pmod{5}$ because when both 23 and 8 are divided by 5, we end up with a remainder of 3. The command `mod` allows us to compute such a remainder:

```
sage: mod(23, 5)
3
sage: mod(8, 5)
3
```

Euler's phi function

Consider all the integers from 1 to 20, inclusive. List all those integers that are coprime to 20. In other words, we want to find those integers n, where $1 \leq n \leq 20$, such that $\gcd(n, 20) = 1$. The latter task can be easily accomplished with a little bit of Sage programming:

```
sage: for n in range(1, 21):
...       if gcd(n, 20) == 1:
...           print n,
...
1 3 7 9 11 13 17 19
```

The above programming statements can be saved to a text file called, say, `/home/mvngu/totient.sage`, organizing it as follows to enhance readability.

```
for n in xrange(1, 21):
    if gcd(n, 20) == 1:
        print n,
```

We refer to `totient.sage` as a Sage script, just as one would refer to a file containing Python code as a Python script. We use 4 space indentations, which is a coding convention in Sage as well as Python programming, instead of tabs.

The command `load` can be used to read the file containing our programming statements into Sage and, upon loading the content of the file, have Sage execute those statements:

```
load("/home/mvngu/totient.sage")
1 3 7 9 11 13 17 19
```

From the latter list, there are 8 integers in the closed interval $[1, 20]$ that are coprime to 20. Without explicitly generating the list

```
1  3  7  9  11  13  17  19
```

how can we compute the number of integers in $[1, 20]$ that are coprime to 20? This is where Euler's phi function comes in handy. Let $n \in \mathbf{Z}$ be positive. Then *Euler's phi function* counts the number of integers a, with $1 \leq a \leq n$, such that $\gcd(a, n) = 1$. This number is denoted by $\varphi(n)$. Euler's phi function is sometimes referred to as Euler's totient function, hence the name `totient.sage` for the above Sage script. The command `euler_phi` implements Euler's phi function. To compute $\varphi(20)$ without explicitly generating the above list, we proceed as follows:

```
sage: euler_phi(20)
8
```

How to keep a secret?

Cryptography is the science (some might say art) of concealing data. Imagine that we are composing a confidential email to someone. Having written the email, we can send it in one of two ways. The first, and usually convenient, way is to simply press the send button and not care about how our email will be delivered. Sending an email in this manner is similar to writing our confidential message on a postcard and post it without enclosing our postcard inside an envelope. Anyone who can access our postcard can see our message. On the other hand, before sending our email, we can scramble the confidential message and then press the send button. Scrambling our message is similar to enclosing our postcard inside an envelope. While not 100% secure, at least we know that anyone wanting to read our postcard has to open the envelope.

In cryptography parlance, our message is called *plaintext*. The process of scrambling our message is referred to as *encryption*. After encrypting our message, the scrambled version is called *ciphertext*. From the ciphertext, we can recover our original unscrambled message via *decryption*. The following figure illustrates the processes of encryption and decryption. A *cryptosystem* is comprised of a pair of related encryption and decryption processes.

```
+ ---------+   encrypt    +------------+   decrypt    +-----------+
| plaintext| -----------> | ciphertext | -----------> | plaintext |
+----------+              +------------+              +-----------+
```

The following table provides a very simple method of scrambling a message written in English and using only upper case letters, excluding punctuation characters.

```
+-------------------------------------------------+
| A   B   C   D   E   F   G   H   I   J   K   L   M  |
| 65  66  67  68  69  70  71  72  73  74  75  76  77 |
+-------------------------------------------------+
| N   O   P   Q   R   S   T   U   V   W   X   Y   Z  |
| 78  79  80  81  82  83  84  85  86  87  88  89  90 |
+-------------------------------------------------+
```

Formally, let

$$\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{Z}\}$$

be the set of capital letters of the English alphabet. Furthermore, let

$$\Phi = \{65, 66, 67, \ldots, 90\}$$

be the American Standard Code for Information Interchange (ASCII) encodings of the upper case English letters. Then the above table explicitly describes the mapping $f : \Sigma \longrightarrow \Phi$. (For those familiar with ASCII, f is actually a common process for *encoding* elements of Σ, rather than a cryptographic "scrambling" process *per se*.) To scramble a message written using the alphabet Σ, we simply replace each capital letter of the message with its corresponding ASCII encoding. However, the scrambling process described in the above table provides, cryptographically speaking, very little to no security at all and we strongly discourage its use in practice.

Keeping a secret with two keys

The Rivest, Shamir, Adleman (RSA) cryptosystem is an example of a *public key cryptosystem*. RSA uses a *public key* to encrypt messages and decryption is performed using a corresponding *private key*. We can distribute our public keys, but for security reasons we should keep our private keys to ourselves. The encryption and decryption processes draw upon techniques from elementary number theory. The algorithm below is adapted from page 165 of [TrappeWashington2006]. It outlines the RSA procedure for encryption and decryption.

1. Choose two primes p and q and let $n = pq$.
2. Let $e \in \mathbf{Z}$ be positive such that $\gcd\big(e, \varphi(n)\big) = 1$.
3. Compute a value for $d \in \mathbf{Z}$ such that $de \equiv 1 \pmod{\varphi(n)}$.
4. Our public key is the pair (n, e) and our private key is the triple (p, q, d).
5. For any non-zero integer $m < n$, encrypt m using $c \equiv m^e \pmod{n}$.
6. Decrypt c using $m \equiv c^d \pmod{n}$.

The next two sections will step through the RSA algorithm, using Sage to generate public and private keys, and perform encryption and decryption based on those keys.

Generating public and private keys

Positive integers of the form $M_m = 2^m - 1$ are called *Mersenne numbers*. If p is prime and $M_p = 2^p - 1$ is also prime, then M_p is called a *Mersenne prime*. For example, 31 is prime and $M_{31} = 2^{31} - 1$ is a Mersenne prime, as can be verified using the command `is_prime(p)`. This command returns `True` if its argument `p` is precisely a prime number; otherwise it returns `False`. By definition, a prime must be a positive integer, hence `is_prime(-2)` returns `False` although we know that 2 is prime. Indeed, the number $M_{61} = 2^{61} - 1$ is also a Mersenne prime. We can use M_{31} and M_{61} to work through step 1 in the RSA algorithm:

```
sage: p = (2^31) - 1
sage: is_prime(p)
True
sage: q = (2^61) - 1
sage: is_prime(q)
True
sage: n = p * q ; n
4951760154835678088235319297
```

A word of warning is in order here. In the above code example, the choice of p and q as Mersenne primes, and with so many digits far apart from each other, is a very bad choice in terms of cryptographic security. However, we shall use the above chosen numeric values for p and q for the remainder of this tutorial, always bearing in mind that they have been chosen for pedagogy purposes only. Refer to [MenezesEtAl1996], [Stinson2006], and [TrappeWashington2006] for in-depth discussions on the security of RSA, or consult other specialized texts.

For step 2, we need to find a positive integer that is coprime to $\varphi(n)$. The set of integers is implemented within the Sage module `sage.rings.integer_ring`. Various operations on integers can be accessed via the `ZZ.*` family of functions. For instance, the command `ZZ.random_element(n)` returns a pseudo-random integer uniformly distributed within the closed interval $[0, n-1]$.

We can compute the value $\varphi(n)$ by calling the sage function `euler_phi(n)`, but for arbitrarily large prime numbers p and q, this can take an enormous amount of time. Indeed, the private key can be quickly deduced from the public key once you know $\varphi(n)$, so it is an important part of the security of the RSA cryptosystem that $\varphi(n)$ cannot be computed in a short time, if only n is known. On the other hand, if the private key is available, we can compute $\varphi(n) = (p-1)(q-1)$ in a very short time.

Using a simple programming loop, we can compute the required value of e as follows:

```
sage: p = (2^31) - 1
sage: q = (2^61) - 1
sage: n = p * q
sage: phi = (p - 1)*(q - 1); phi
4951760152529835076874141700
sage: e = ZZ.random_element(phi)
sage: while gcd(e, phi) != 1:
...       e = ZZ.random_element(phi)
...
sage: e  # random
185056762330061596630395 4877
sage: e < n
True
```

As `e` is a pseudo-random integer, its numeric value changes after each execution of `e = ZZ.random_element(phi)`.

To calculate a value for `d` in step 3 of the RSA algorithm, we use the extended Euclidean algorithm. By definition of congruence, $de \equiv 1 \pmod{\varphi(n)}$ is equivalent to

$$de - k \cdot \varphi(n) = 1$$

where $k \in \mathbf{Z}$. From steps 1 and 2, we already know the numeric values of e and $\varphi(n)$. The extended Euclidean algorithm allows us to compute d and $-k$. In Sage, this can be accomplished via the command `xgcd`. Given two integers x and y, `xgcd(x, y)` returns a 3-tuple `(g, s, t)` that satisfies the Bézout identity $g = \gcd(x, y) = sx + ty$. Having computed a value for `d`, we then use the command `mod(d*e, phi)` to check that `d*e` is indeed congruent to 1 modulo `phi`.

```
sage: n = 4951760154835678088235319297
sage: e = 185056762330061596630395 4877
sage: phi = 4951760152529835076874141700
sage: bezout = xgcd(e, phi); bezout  # random
(1, 4460824882019967172592779313, -166709570851537792508703 3035)
sage: d = Integer(mod(bezout[1], phi)) ; d  # random
4460824882019967172592779313
sage: mod(d * e, phi)
1
```

Thus, our RSA public key is

$$(n, e) = (4951760154835678088235319297,\ 185056762330061596630395 4877)$$

and our corresponding private key is

$$(p, q, d) = (2147483647,\ 2305843009213693951,\ 4460824882019967172592779313)$$

Encryption and decryption

Suppose we want to scramble the message `HELLOWORLD` using RSA encryption. From the above ASCII table, our message maps to integers of the ASCII encodings as given below.

```
+----------------------------------------+
| H   E   L   L   O   W   O   R   L   D  |
| 72  69  76  76  79  87  79  82  76  68 |
+----------------------------------------+
```

Concatenating all the integers in the last table, our message can be represented by the integer

$$m = 72697676798779827668$$

There are other more cryptographically secure means for representing our message as an integer. The above process is used for demonstration purposes only and we strongly discourage its use in practice. In Sage, we can obtain an integer representation of our message as follows:

```
sage: m = "HELLOWORLD"
sage: m = map(ord, m); m
[72, 69, 76, 76, 79, 87, 79, 82, 76, 68]
sage: m = ZZ(list(reversed(m)), 100) ; m
72697676798779827668
```

To encrypt our message, we raise m to the power of e and reduce the result modulo n. The command `mod(a^b, n)` first computes `a^b` and then reduces the result modulo `n`. If the exponent `b` is a "large" integer, say with more than 20 digits, then performing modular exponentiation in this naive manner takes quite some time. Brute force (or naive) modular exponentiation is inefficient and, when performed using a computer, can quickly consume a huge quantity of the computer's memory or result in overflow messages. For instance, if we perform naive modular exponentiation using the command `mod(m^e, n)`, where `m`, `n` and `e` are as given above, we would get an error message similar to the following:

```
mod(m^e, n)
Traceback (most recent call last)
/home/mvngu/<ipython console> in <module>()
/home/mvngu/usr/bin/sage-3.1.4/local/lib/python2.5/site-packages/sage/rings/integer.so
in sage.rings.integer.Integer.__pow__ (sage/rings/integer.c:9650)()
RuntimeError: exponent must be at most 2147483647
```

There is a trick to efficiently perform modular exponentiation, called the method of repeated squaring, cf. page 879 of [CormenEtAl2001]. Suppose we want to compute $a^b \mod n$. First, let $d := 1$ and obtain the binary representation of b, say $(b_1, b_2, \ldots, b_k)$ where each $b_i \in \mathbf{Z}/2\mathbf{Z}$. For $i := 1, \ldots, k$, let $d := d^2 \mod n$ and if $b_i = 1$ then let $d := da \mod n$. This algorithm is implemented in the function `power_mod`. We now use the function `power_mod` to encrypt our message:

```
sage: m = 72697676798779827668
sage: e = 18505676233006159663039548 77
sage: n = 49517601548356780882353192 97
sage: c = power_mod(m, e, n); c
63091363257752005841552109 0
```

Thus $c = 630913632577520058415521090$ is the ciphertext. To recover our plaintext, we raise `c` to the power of `d` and reduce the result modulo `n`. Again, we use modular exponentiation via repeated squaring in the decryption process:

```
sage: m = 72697676798779827668
sage: c = 630913632577520058415521090
sage: d = 44608248820199671725927793 13
sage: n = 49517601548356780882353192 97
sage: power_mod(c, d, n)
72697676798779827668
sage: power_mod(c, d, n) == m
True
```

Notice in the last output that the value 72697676798779827668 is the same as the integer that represents our original message. Hence we have recovered our plaintext.

Acknowledgements

1. 2009-07-25: Ron Evans (Department of Mathematics, UCSD) reported a typo in the definition of greatest common divisors. The revised definition incorporates his suggestions.
2. 2008-11-04: Martin Albrecht (Information Security Group, Royal Holloway, University of London), John Cremona (Mathematics Institute, University of Warwick) and William Stein (Department of Mathematics, University of Washington) reviewed this tutorial. Many of their invaluable suggestions have been incorporated into this document.

Bibliography

6.1.8 Tutorial: Programming in Python and Sage

Author: Florent Hivert <florent.hivert@univ-rouen.fr>, Franco Saliola <saliola@gmail.com>, et al.

This tutorial is an introduction to basic programming in Python and Sage, for readers with elementary notions of programming but not familiar with the Python language. It is far from exhaustive. For a more complete tutorial, have a look at the Python Tutorial. Also Python's documentation and in particular the standard library can be useful.

A *more advanced tutorial* presents the notions of objects and classes in Python.

Here are further resources to learn Python:

- Learn Python in 10 minutes ou en français Python en 10 minutes
- Dive into Python is a Python book for experienced programmers. Also available in other languages.
- Discover Python is a series of articles published in IBM's developerWorks technical resource center.

Data structures

In Python, *typing is dynamic*; there is no such thing as declaring variables. The function `type()` returns the type of an object `obj`. To convert an object to a type `typ` just write `typ(obj)` as in `int("123")`. The command `isinstance(ex, typ)` returns whether the expression `ex` is of type `typ`. Specifically, any value is *an instance of a class* and there is no difference between classes and types.

The symbol = denotes the affectation to a variable; it should not be confused with == which denotes mathematical equality. Inequality is `!=`.

The *standard types* are `bool`, `int`, `list`, `tuple`, `set`, `dict`, `str`.

- The type `bool` (*booleans*) has two values: `True` and `False`. The boolean operators are denoted by their names `or`, `and`, `not`.

- The Python types `int` and `long` are used to represent integers of limited size. To handle arbitrary large integers with exact arithmetic, Sage uses its own type named `Integer`.
- A *list* is a data structure which groups values. It is constructed using brackets as in `[1, 3, 4]`. The `range()` function creates integer lists. One can also create lists using *list comprehension*:

  ```
  [ <expr> for <name> in <iterable> (if <condition>) ]
  ```

  For example:

  ```
  sage: [ i^2 for i in range(10) if i % 2 == 0 ]
  [0, 4, 16, 36, 64]
  ```

- A *tuple* is very similar to a list; it is constructed using parentheses. The empty tuple is obtained by `()` or by the constructor `tuple`. If there is only one element, one has to write `(a,)`. A tuple is *immutable* (one cannot change it) but it is *hashable* (see below). One can also create tuples using comprehensions:

  ```
  sage: tuple(i^2 for i in range(10) if i % 2 == 0)
  (0, 4, 16, 36, 64)
  ```

- A *set* is a data structure which contains values without multiplicities or order. One creates it from a list (or any iterable) with the constructor `set`. The elements of a set must be hashable:

  ```
  sage: set([2,2,1,4,5])
  set([1, 2, 4, 5])

  sage: set([ [1], [2] ])
  Traceback (most recent call last):
  ...
  TypeError: unhashable type: 'list'
  ```

- A *dictionary* is an association table, which associates values to keys. Keys must be hashable. One creates dictionaries using the constructor `dict`, or using the syntax:

  ```
  {key1 : value1, key2 : value2 ...}
  ```

  For example:

  ```
  sage: age = {'toto' : 8, 'mom' : 27}; age
  {'toto': 8, 'mom': 27}
  ```

- Quotes (simple `' '` or double `" "`) enclose *character strings*. One can concatenate them using `+`.
- For lists, tuples, strings, and dictionaries, the *indexing operator* is written `l[i]`. For lists, tuples, and strings one can also uses *slices* as `l[:]`, `l[:b]`, `l[a:]`, or `l[a:b]`. Negative indices start from the end.
- The `len()` function returns the number of elements of a list, a tuple, a set, a string, or a dictionary. One writes `x in C` to tests whether `x` is in `C`.
- Finally there is a special value called `None` to denote the absence of a value.

Control structures

In Python, there is no keyword for the beginning and the end of an instructions block. Blocks are delimited solely by means of indentation. Most of the time a new block is introduced by `:`. Python has the following control structures:

- Conditional instruction:

  ```
  if <condition>:
      <instruction sequence>
  [elif <condition>:
  ```

```
    <instruction sequence>]*
[else:
    <instruction sequence>]
```

- Inside expression exclusively, one can write:

```
<value> if <condition> else <value>
```

- Iterative instructions:

```
for <name> in <iterable>:
    <instruction sequence>
[else:
    <instruction sequence>]

while <condition>:
    <instruction sequence>
[else:
    <instruction sequence>]
```

  The `else` block is executed at the end of the loop if the loop is ended normally, that is neither by a `break` nor an exception.

- In a loop, `continue` jumps to the next iteration.
- An iterable is an object which can be iterated through. Iterable types include lists, tuples, dictionaries, and strings.
- An error (also called exception) is raised by:

```
raise <ErrorType> [, error message]
```

  Usual errors include `ValueError` and `TypeError`.

Functions

Note: Python functions vs. mathematical functions

In what follows, we deal with *functions* is the sense of *programming languages*. Mathematical functions, as manipulated in calculus, are handled by Sage in a different way. In particular it doesn't make sense to do mathematical manipulation such as additions or derivations on Python functions.

One defines a function using the keyword `def` as:

```
def <name>(<argument list>):
     <instruction sequence>
```

The result of the function is given by the instruction `return`. Very short functions can be created anonymously using `lambda` (remark that there is no instruction `return` here):

```
lambda <arguments>: <expression>
```

Note: Functional programming

Functions are objects as any other objects. One can assign them to variables or return them. For details, see the tutorial on *Functional Programming for Mathematicians*.

Exercises

Lists

Creating Lists I: [Square brackets] Example:

```
sage: L = [3, Permutation([5,1,4,2,3]), 17, 17, 3, 51]
sage: L
[3, [5, 1, 4, 2, 3], 17, 17, 3, 51]
```

Exercise: Create the list $[63, 12, -10, \text{“a”}, 12]$, assign it to the variable `L`, and print the list.

```
sage: # edit here
```

Exercise: Create the empty list (you will often need to do this).

```
sage: # edit here
```

Creating Lists II: range The `range()` function provides an easy way to construct a list of integers. Here is the documentation of the `range()` function:

```
range([start,] stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
When step is given, it specifies the increment (or decrement). For
example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
These are exactly the valid indices for a list of 4 elements.
```

Exercise: Use `range()` to construct the list $[1, 2, \ldots, 50]$.

```
sage: # edit here
```

Exercise: Use `range()` to construct the list of *even* numbers between 1 and 100 (including 100).

```
sage: # edit here
```

Exercise: The `step` argument for the `range()` command can be negative. Use range to construct the list $[10, 7, 4, 1, -2]$.

```
sage: # edit here
```

See Also:

- `xrange()`: returns an iterator rather than building a list.
- `srange()`: like range but with Sage integers; see below.
- `sxrange()`: like xrange but with Sage integers.

Creating Lists III: list comprehensions *List comprehensions* provide a concise way to create lists from other lists (or other data types).

Example We already know how to create the list $[1, 2, \ldots, 16]$:

```
sage: range(1,17)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

Using a *list comprehension*, we can now create the list $[1^2, 2^2, 3^2, \ldots, 16^2]$ as follows:

```
sage: [i^2 for i in range(1,17)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256]
```

```
sage: sum([i^2 for i in range(1,17)])
1496
```

Exercise: [Project Euler, Problem 6]

The sum of the squares of the first ten natural numbers is

$$(1^2 + 2^2 + ... + 10^2) = 385$$

The square of the sum of the first ten natural numbers is

$$(1 + 2 + ... + 10)^2 = 55^2 = 3025$$

Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is

$$3025 - 385 = 2640$$

Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

```
sage: # edit here
```

```
sage: # edit here
```

```
sage: # edit here
```

Filtering lists with a list comprehension A list can be *filtered* using a list comprehension.

Example: To create a list of the squares of the prime numbers between 1 and 100, we use a list comprehension as follows.

```
sage: [p^2 for p in [1,2,..,100] if is_prime(p)]
[4, 9, 25, 49, 121, 169, 289, 361, 529, 841, 961, 1369, 1681, 1849, 2209, 2809, 3481, 3721, 4489, 504
```

Exercise: Use a *list comprehension* to list all the natural numbers below 20 that are multiples of 3 or 5. Hint:

- To get the remainder of 7 divided by 3 use `7%3`.
- To test for equality use two equal signs (==); for example, `3 == 7`.

```
sage: # edit here
```

Project Euler, Problem 1: Find the sum of all the multiples of 3 or 5 below 1000.

```
sage: # edit here
```

Nested list comprehensions List comprehensions can be nested!

Examples:

```
sage: [(x,y) for x in range(5) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (3, 0), (3, 1), (3, 2), (4,
```

```
sage: [[i^j for j in range(1,4)] for i in range(6)]
[[0, 0, 0], [1, 1, 1], [2, 4, 8], [3, 9, 27], [4, 16, 64], [5, 25, 125]]
```

```
sage: matrix([[i^j for j in range(1,4)] for i in range(6)])
[  0   0   0]
[  1   1   1]
[  2   4   8]
[  3   9  27]
[  4  16  64]
[  5  25 125]
```

Exercise:

1. A *Pythagorean triple* is a triple (x, y, z) of *positive* integers satisfying $x^2 + y^2 = z^2$. The Pythagorean triples whose components are at most 10 are:

$$[(3, 4, 5), (4, 3, 5), (6, 8, 10), (8, 6, 10)] .$$

   Using a filtered list comprehension, construct the list of Pythagorean triples whose components are at most 50:

   ```
   sage: # edit here
   ```

   ```
   sage: # edit here
   ```

2. Project Euler, Problem 9: There exists exactly one Pythagorean triple for which $a + b + c = 1000$. Find the product abc:

   ```
   sage: # edit here
   ```

Accessing individual elements of lists To access an element of the list `L`, use the syntax `L[i]`, where i is the index of the item.

Exercise:

1. Construct the list `L = [1,2,3,4,3,5,6]`. What is `L[3]`?

   ```
   sage: # edit here
   ```

2. What is `L[1]`?

   ```
   sage: # edit here
   ```

3. What is the index of the first element of `L`?

   ```
   sage: # edit here
   ```

4. What is `L[-1]`? What is `L[-2]`?

```
sage: # edit here
```

5. What is `L.index(2)`? What is `L.index(3)`?

```
sage: # edit here
```

Modifying lists: changing an element in a list To change the item in position `i` of a list `L`:

```
sage: L = ["a", 4, 1, 8]
sage: L
['a', 4, 1, 8]

sage: L[2] = 0
sage: L
['a', 4, 0, 8]
```

Modifying lists: append and extend To *append* an object to a list:

```
sage: L = ["a", 4, 1, 8]
sage: L
['a', 4, 1, 8]

sage: L.append(17)
sage: L
['a', 4, 1, 8, 17]
```

To *extend* a list by another list:

```
sage: L1 = [1,2,3]
sage: L2 = [7,8,9,0]
sage: print L1
[1, 2, 3]
sage: print L2
[7, 8, 9, 0]

sage: L1.extend(L2)
sage: L1
[1, 2, 3, 7, 8, 9, 0]
```

Modifying lists: reverse, sort, ...

```
sage: L = [4,2,5,1,3]
sage: L
[4, 2, 5, 1, 3]

sage: L.reverse()
sage: L
[3, 1, 5, 2, 4]

sage: L.sort()
sage: L
[1, 2, 3, 4, 5]
```

```
sage: L = [3,1,6,4]
sage: sorted(L)
[1, 3, 4, 6]

sage: L
[3, 1, 6, 4]
```

Concatenating Lists To concatenate two lists, add them with the operator +. This is not a commutative operation!

```
sage: L1 = [1,2,3]
sage: L2 = [7,8,9,0]
sage: L1 + L2
[1, 2, 3, 7, 8, 9, 0]
```

Slicing Lists You can slice a list using the syntax `L[start : stop : step]`. This will return a sublist of `L`.

Exercise: Below are some examples of slicing lists. Try to guess what the output will be before evaluating the cell:

```
sage: L = range(20)
sage: L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

sage: L[3:15]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

sage: L[3:15:2]
[3, 5, 7, 9, 11, 13]

sage: L[15:3:-1]
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4]

sage: L[:4]
[0, 1, 2, 3]

sage: L[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

sage: L[::-1]
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Exercise (Advanced): The following function combines a loop with some of the list operations above. What does the function do?

```
sage: def f(number_of_iterations):
...       L = [1]
...       for n in range(2, number_of_iterations):
...           L = [sum(L[:i]) for i in range(n-1, -1, -1)]
...       return numerical_approx(2*L[0]*len(L)/sum(L), digits=50)

sage: # edit here
```

Tuples

A *tuple* is an *immutable* list. That is, it cannot be changed once it is created. This can be useful for code safety and foremost because it makes tuple *hashable*. To create a tuple, use parentheses instead of brackets:

```
sage: t = (3, 5, [3,1], (17,[2,3],17), 4)
sage: t
(3, 5, [3, 1], (17, [2, 3], 17), 4)
```

To create a singleton tuple, a comma is required to resolve the ambiguity:

```
sage: (1)
1
sage: (1,)
(1,)
```

We can create a tuple from a list, and vice-versa.

```
sage: tuple(range(5))
(0, 1, 2, 3, 4)
```

```
sage: list(t)
[3, 5, [3, 1], (17, [2, 3], 17), 4]
```

Tuples behave like lists in many respects:

Operation	Syntax for lists	Syntax for tuples
Accessing a letter	`list[3]`	`tuple[3]`
Concatenation	`list1 + list2`	`tuple1 + tuple2`
Slicing	`list[3:17:2]`	`tuple[3:17:2]`
A reversed copy	`list[::-1]`	`tuple[::-1]`
Length	`len(list)`	`len(tuple)`

Trying to modify a tuple will fail:

```
sage: t = (5, 'a', 6/5)
sage: t
(5, 'a', 6/5)
```

```
sage: t[1] = 'b'
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Generators

"Tuple-comprehensions" do not exist. Instead, the syntax produces something called a generator. A generator allows you to process a sequence of items one at a time. Each item is created when it is needed, and then forgotten. This can be very efficient if we only need to use each item once.

```
sage: (i^2 for i in range(5))
<generator object <genexpr> at 0x...>
```

```
sage: g = (i^2 for i in range(5))
sage: g[0]
```

```
Traceback (most recent call last):
...
TypeError: 'generator' object has no attribute '__getitem__'
```

```
sage: [x for x in g]
[0, 1, 4, 9, 16]
```

g is now empty.

```
sage: [x for x in g]
[]
```

A nice 'pythonic' trick is to use generators as argument of functions. We do *not* need double parentheses for this:

```
sage: sum( i^2 for i in srange(100001) )
333338333350000
```

Dictionaries

A *dictionary* is another built-in data type. Unlike lists, which are indexed by a range of numbers starting at 0, dictionaries are indexed by *keys*, which can be any immutable objects. Strings and numbers can always be keys (because they are immutable). Dictionaries are sometimes called "associative arrays" in other programming languages.

There are several ways to define dictionaries. One method is to use braces, {}, with comma-separated entries given in the form *key:value*:

```
sage: d = {3:17, "key":[4,1,5,2,3], (3,1,2):"goo", 3/2 : 17}
sage: d
{3/2: 17, 3: 17, (3, 1, 2): 'goo', 'key': [4, 1, 5, 2, 3]}
```

A second method is to use the constructor `dict` which admits a list (or actually any iterable) of 2-tuples *(key, value)*:

```
sage: dd = dict((i,i^2) for i in xrange(10))
sage: dd
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Dictionaries behave as lists and tuples for several important operations.

Operation	Syntax for lists	Syntax for dictionaries
Accessing elements	`list[3]`	`D["key"]`
Length	`len(list)`	`len(D)`
Modifying	`L[3] = 17`	`D["key"] = 17`
Deleting items	`del L[3]`	`del D["key"]`

```
sage: d[10]='a'
sage: d
{3/2: 17, 10: 'a', 3: 17, (3, 1, 2): 'goo', 'key': [4, 1, 5, 2, 3]}
```

A dictionary can have the same value multiple times, but each key must only appear once and must be immutable:

```
sage: d = {3: 14, 4: 14}
sage: d
{3: 14, 4: 14}
```

```
sage: d = {3: 13, 3: 14}
sage: d
{3: 14}
```

```
sage: d = {[1,2,3] : 12}
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

Another way to add items to a dictionary is with the `update()` method which updates the dictionary from another dictionary:

```
sage: d = {}
sage: d
{}
```

```
sage: d.update( {10 : 'newvalue', 20: 'newervalue', 3: 14, 'a':[1,2,3]} )
sage: d
{'a': [1, 2, 3], 10: 'newvalue', 3: 14, 20: 'newervalue'}
```

We can iterate through the *keys*, or *values*, or both, of a dictionary:

```
sage: d = {10 : 'newvalue', 20: 'newervalue', 3: 14, 'a':[1,2,3]}
```

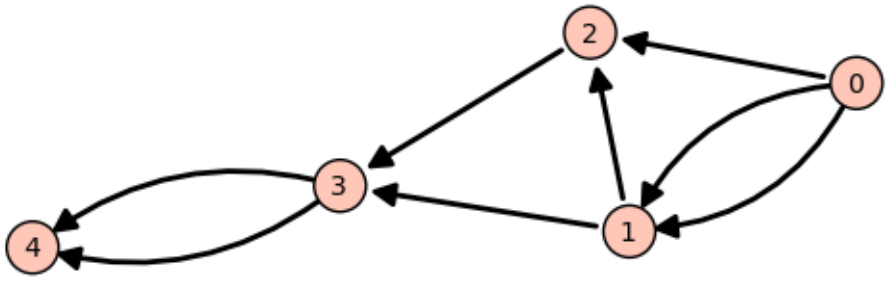
```
sage: [key for key in d]
['a', 10, 3, 20]
```

```
sage: [key for key in d.iterkeys()]
['a', 10, 3, 20]
```

```
sage: [value for value in d.itervalues()]
[[1, 2, 3], 'newvalue', 14, 'newervalue']
```

```
sage: [(key, value) for key, value in d.iteritems()]
[('a', [1, 2, 3]), (10, 'newvalue'), (3, 14), (20, 'newervalue')]
```

Exercise: Consider the following directed graph.



Create a dictionary whose keys are the vertices of the above directed graph, and whose values are the lists of the vertices that it points to. For instance, the vertex 1 points to the vertices 2 and 3, so the dictionary will look like:

```
d = { ..., 1:[2,3], ... }

sage: # edit here
```

Then try:

```
sage: g = DiGraph(d)
sage: g.plot()
```

Using Sage types: The srange command

Example: Construct a 3×3 matrix whose (i, j) entry is the rational number $\frac{i}{j}$. The integers generated by `range()` are Python `int`'s. As a consequence, dividing them does euclidean division:

```
sage: matrix([[ i/j for j in range(1,4)] for i in range(1,4)])
[1 0 0]
[2 1 0]
[3 1 1]
```

Whereas dividing a Sage `Integer` by a Sage `Integer` produces a rational number:

```
sage: matrix([[ i/j for j in srange(1,4)] for i in srange(1,4)])
[  1 1/2 1/3]
[  2   1 2/3]
[  3 3/2   1]
```

Modifying lists has consequences!

Try to predict the results of the following commands:

```
sage: a = [1, 2, 3]
sage: L = [a, a, a]
sage: L
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]

sage: a.append(4)
sage: L
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
```

Now try these:

```
sage: a = [1, 2, 3]
sage: L = [a, a, a]
sage: L
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]

sage: a = [1, 2, 3, 4]
sage: L
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]

sage: L[0].append(4)
sage: L
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
```

This is known as the *reference effect*. You can use the command `deepcopy()` to avoid this effect:

```
sage: a = [1,2,3]
sage: L = [deepcopy(a), deepcopy(a)]
sage: L
[[1, 2, 3], [1, 2, 3]]

sage: a.append(4)
sage: L
[[1, 2, 3], [1, 2, 3]]
```

The same effect occurs with dictionaries:

```
sage: d = {1:'a', 2:'b', 3:'c'}
sage: dd = d
sage: d.update( { 4:'d' } )
sage: dd
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

Loops and Functions

For more verbose explanation of what's going on here, a good place to look at is the following section of the Python tutorial: http://docs.python.org/tutorial/controlflow.html

While Loops

While loops tend not to be used nearly as much as *for* loops in Python code:

```
sage: i = 0
sage: while i < 10:
...       print i
...       i += 1
0
1
2
3
4
5
6
7
8
9

sage: i = 0
sage: while i < 10:
...       if i % 2 == 1:
...           i += 1
...           continue
...       print i
...       i += 1
0
2
4
6
8
```

Note that the truth value of the clause expression in the *while* loop is evaluated using `bool`:

```
sage: bool(True)
True

sage: bool('a')
True

sage: bool(1)
True

sage: bool(0)
False

sage: i = 4
sage: while i:
...       print i
...       i -= 1
4
3
2
1
```

For Loops

Here is a basic *for* loop iterating over all of the elements in the list `l`:

```
sage: l = ['a', 'b', 'c']
sage: for letter in l:
...       print letter
a
b
c
```

The `range()` function is very useful when you want to generate arithmetic progressions to loop over. Note that the end point is never included:

```
sage: range?

sage: range(4)
[0, 1, 2, 3]

sage: range(1, 5)
[1, 2, 3, 4]

sage: range(1, 11, 2)
[1, 3, 5, 7, 9]

sage: range(10, 0, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

sage: for i in range(4):
...       print i, i*i
0 0
```

```
1 1
2 4
3 9
```

You can use the *continue* keyword to immediately go to the next item in the loop:

```
sage: for i in range(10):
...       if i % 2 == 0:
...           continue
...       print i
1
3
5
7
9
```

If you want to break out of the loop, use the *break* keyword:

```
sage: for i in range(10):
...       if i % 2 == 0:
...           continue
...       if i == 7:
...           break
...       print i
1
3
5
```

If you need to keep track of both the position in the list and its value, one (not so elegant) way would be to do the following:

```
sage: l = ['a', 'b', 'c']
...   for i in range(len(l)):
...       print i, l[i]
```

It's cleaner to use `enumerate()` which provides the index as well as the value:

```
sage: l = ['a', 'b', 'c']
...   for i, letter in enumerate(l):
...       print i, letter
```

You could get a similar result to the result of the `enumerate()` function by using `zip()` to zip two lists together:

```
sage: l = ['a', 'b', 'c']
...   for i, letter in zip(range(len(l)), l):
...       print i, letter
```

For loops work using Python's iterator protocol. This allows all sorts of different objects to be looped over. For example:

```
sage: for i in GF(5):
...       print i, i*i
0 0
1 1
2 4
3 4
4 1
```

How does this work?

```
sage: it = iter(GF(5)); it
<generator object __iter__ at 0x...>

sage: it.next()
0

sage: it.next()
1

sage: it.next()
2

sage: it.next()
3

sage: it.next()
4

sage: it.next()
Traceback (most recent call last):
...
StopIteration


sage: R = GF(5)
...   R.__iter__??
```

The command *yield* provides a very convenient way to produce iterators. We'll see more about it in a bit.

Exercises For each of the following sets, compute the list of its elements and their sum. Use two different ways, if possible: with a loop, and using a list comprehension.

1. The first n terms of the harmonic series:

$$\sum_{i=1}^{n} \frac{1}{i}$$

   ```
   sage: # edit here
   ```

2. The odd integers between 1 and n:

   ```
   sage: # edit here
   ```

3. The first n odd integers:

   ```
   sage: # edit here
   ```

4. The integers between 1 and n that are neither divisible by 2 nor by 3 nor by 5:

   ```
   sage: # edit here
   ```

5. The first n integers between 1 and n that are neither divisible by 2 nor by 3 nor by 5:

```
sage: # edit here
```

Functions

Functions are defined using the *def* statement, and values are returned using the *return* keyword:

```
sage: def f(x):
...       return x*x

sage: f(2)
4
```

Functions can be recursive:

```
sage: def fib(n):
...       if n <= 1:
...           return 1
...       else:
...           return fib(n-1) + fib(n-2)

sage: [fib(i) for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Functions are first class objects like any other. For example, they can be passed in as arguments to other functions:

```
sage: f
<function f at 0x...>

sage: def compose(f, x, n):    # computes f(f(...f(x)))
...       for i in range(n):
...           x = f(x)         # this change is local to this function call!
...       return x

sage: compose(f, 2, 3)
256

sage: def add_one(x):
...       return x + 1

sage: compose(add_one, 2, 3)
5
```

You can give default values for arguments in functions:

```
sage: def add_n(x, n=1):
...       return x + n

sage: add_n(4)
5

sage: add_n(4, n=100)
104
```

```
sage: add_n(4, 1000)
1004
```

You can return multiple values from a function:

```
sage: def g(x):
...       return x, x*x

sage: g(2)
(2, 4)

sage: type(g)
<type 'function'>

sage: a,b = g(100)

sage: a
100

sage: b
10000
```

You can also take a variable number of arguments and keyword arguments in a function:

```
sage: def h(*args, **kwds):
...       print type(args), args
...       print type(kwds), kwds

sage: h(1,2,3,n=4)
<type 'tuple'> (1, 2, 3)
<type 'dict'> {'n': 4}
```

Let's use the *yield* instruction to make a generator for the Fibonacci numbers up to n:

```
sage: def fib_gen(n):
...       if n < 1:
...           return
...       a = b = 1
...       yield b
...       while b < n:
...           yield b
...           a, b = b, b+a

sage: for i in fib_gen(50):
...       print i
1
1
2
3
5
8
13
21
34
```

Exercises

1. Write a function `is_even` which returns `True` if `n` is even and `False` otherwise.
2. Write a function `every_other` which takes a list `l` as input and returns a list containing every other element of `l`.
3. Write a generator `every_other` which takes an iterable `l` as input, and returns every other element of `l`, one after the other.
4. Write a function which computes the n-th Fibonacci number. Try to improve performance.

Todo

- Definition of `hashable`
- Introduction to the debugger.

6.1.9 Tutorial: Comprehensions, Iterators, and Iterables

Author: Florent Hivert <florent.hivert@univ-rouen.fr> and Nicolas M. Thiéry <nthiery at users.sf.net>

List comprehensions

List comprehensions are a very handy way to construct lists in Python. You can use either of the following idioms:

```
[ <expr> for <name> in <iterable> ]
[ <expr> for <name> in <iterable> if <condition> ]
```

For example, here are some lists of squares:

```
sage: [ i^2 for i in [1, 3, 7] ]
[1, 9, 49]
sage: [ i^2 for i in range(1,10) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
sage: [ i^2 for i in range(1,10) if i % 2 == 1]
[1, 9, 25, 49, 81]
```

And a variant on the latter:

```
sage: [i^2 if i % 2 == 1 else 2 for i in range(10)]
[2, 1, 2, 9, 2, 25, 2, 49, 2, 81]
```

Exercises

1. Construct the list of the squares of the prime integers between 1 and 10:

   ```
   sage: # edit here
   ```

2. Construct the list of the perfect squares less than 100 (hint: use `srange()` to get a list of Sage integers together with the method `i.sqrtrem()`):

   ```
   sage: # edit here
   ```

One can use more than one iterable in a list comprehension:

```
sage: [ (i,j) for i in range(1,6) for j in range(1,i) ]
[(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3), (5, 1), (5, 2), (5, 3), (5, 4)]
```

Warning: Mind the order of the nested loop in the previous expression.

If instead one wants to build a list of lists, one can use nested lists as in:

```
sage: [ [ binomial(n, i) for i in range(n+1) ] for n in range(10) ]
[[1],
[1, 1],
[1, 2, 1],
[1, 3, 3, 1],
[1, 4, 6, 4, 1],
[1, 5, 10, 10, 5, 1],
[1, 6, 15, 20, 15, 6, 1],
[1, 7, 21, 35, 35, 21, 7, 1],
[1, 8, 28, 56, 70, 56, 28, 8, 1],
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

Exercises

1. Compute the list of pairs (i, j) of non negative integers such that `i` is at most 5, `j` is at most 8, and `i` and `j` are co-prime:

   ```
   sage: # edit here
   ```

2. Compute the same list for $i < j < 10$:

   ```
   sage: # edit here
   ```

Iterators

Definition

To build a comprehension, Python actually uses an *iterator*. This is a device which runs through a bunch of objects, returning one at each call to the `next` method. Iterators are built using parentheses:

```
sage: it = (binomial(8, i) for i in range(9))
sage: it.next()
1

sage: it.next()
8
sage: it.next()
28
sage: it.next()
56
```

You can get the list of the results that are not yet *consumed*:

```
sage: list(it)
[70, 56, 28, 8, 1]
```

Asking for more elements triggers a `StopIteration` exception:

```
sage: it.next()
Traceback (most recent call last):
...
StopIteration
```

An iterator can be used as argument for a function. The two following idioms give the same results; however, the second idiom is much more memory efficient (for large examples) as it does not expand any list in memory:

```
sage: sum( [ binomial(8, i) for i in range(9) ] )
256
sage: sum( binomial(8, i) for i in xrange(9) )
256
```

Exercises

1. Compute the sum of $\binom{10}{i}$ for all even i:

   ```
   sage: # edit here
   ```

2. Compute the sum of the gcd's of all co-prime numbers i, j for $i < j < 10$:

   ```
   sage: # edit here
   ```

Typical usage of iterators

Iterators are very handy with the functions `all()`, `any()`, and `exists()`:

```
sage: all([True, True, True, True])
True
sage: all([True, False, True, True])
False

sage: any([False, False, False, False])
False
sage: any([False, False, True, False])
True
```

Let's check that all the prime numbers larger than 2 are odd:

```
sage: all( is_odd(p) for p in range(1,100) if is_prime(p) and p>2 )
True
```

It is well know that if `2^p-1` is prime then `p` is prime:

```
sage: def mersenne(p): return 2^p -1
sage: [ is_prime(p) for p in range(20) if is_prime(mersenne(p)) ]
[True, True, True, True, True, True, True]
```

The converse is not true:

```
sage: all( is_prime(mersenne(p)) for p in range(1000) if is_prime(p) )
False
```

Using a list would be much slower here:

```
sage: %time all( is_prime(mersenne(p)) for p in range(1000) if is_prime(p) )     # not tested
CPU times: user 0.00 s, sys: 0.00 s, total: 0.00 s
Wall time: 0.00 s
False
sage: %time all( [ is_prime(mersenne(p)) for p in range(1000) if is_prime(p)] ) # not tested
CPU times: user 0.72 s, sys: 0.00 s, total: 0.73 s
Wall time: 0.73 s
False
```

You can get the counterexample using `exists()`. It takes two arguments: an iterator and a function which tests the property that should hold:

```
sage: exists( (p for p in range(1000) if is_prime(p)), lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

An alternative way to achieve this is:

```
sage: counter_examples = (p for p in range(1000) if is_prime(p) and not is_prime(mersenne(p)))
sage: counter_examples.next()
11
```

Exercises

1. Build the list $\{i^3 \mid -10 < i < 10\}$. Can you find two of those cubes u and v such that $u + v = 218$?

   ```
   sage: # edit here
   ```

itertools

At its name suggests `itertools` is a module which defines several handy tools for manipulating iterators:

```
sage: l = [3, 234, 12, 53, 23]
sage: [(i, l[i]) for i in range(len(l))]
[(0, 3), (1, 234), (2, 12), (3, 53), (4, 23)]
```

The same results can be obtained using `enumerate()`:

```
sage: list(enumerate(l))
[(0, 3), (1, 234), (2, 12), (3, 53), (4, 23)]
```

Here is the analogue of list slicing:

```
sage: list(Permutations(3))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
sage: list(Permutations(3))[1:4]
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]

sage: import itertools
sage: list(itertools.islice(Permutations(3), 1, 4))
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]
```

The functions `map()` and `filter()` also have an analogue:

```
sage: list(itertools.imap(lambda z: z.cycle_type(), Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

```
sage: list(itertools.ifilter(lambda z: z.has_pattern([1,2]), Permutations(3)))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]
```

Exercises

1. Define an iterator for the i-th prime for $5 < i < 10$:

   ```
   sage: # edit here
   ```

Defining new iterators

One can very easily write new iterators using the keyword `yield`. The following function does nothing interesting beyond demonstrating the use of `yield`:

```
sage: def f(n):
...       for i in range(n):
...           yield i
sage: [ u for u in f(5) ]
[0, 1, 2, 3, 4]
```

Iterators can be recursive:

```
sage: def words(alphabet,l):
...       if l == 0:
...           yield []
...       else:
...           for word in words(alphabet, l-1):
...               for a in alphabet:
...                   yield word + [a]

sage: [ w for w in words(['a','b','c'], 3) ]
[['a', 'a', 'a'], ['a', 'a', 'b'], ['a', 'a', 'c'], ['a', 'b', 'a'], ['a', 'b', 'b'], ['a', 'b', 'c']
sage: sum(1 for w in words(['a','b','c'], 3))
27
```

Here is another recursive iterator:

```
sage: def dyck_words(l):
...       if l==0:
...           yield ''
...       else:
...           for k in range(l):
...               for w1 in dyck_words(k):
...                   for w2 in dyck_words(l-k-1):
...                       yield '('+w1+')'+w2

sage: list(dyck_words(4))
['()()()()',
'()()(())',
'()(())()',
'()((()))',
'(())()()',
'(())(())',
'((())())',
```

```
'((()))()',
'(()()())',
'(()(()))',
'((())())',
'((()()))',
'(((())))']

sage: sum(1 for w in dyck_words(5))
42
```

Exercises

1. Write an iterator with two parameters n, l iterating through the set of nondecreasing lists of integers smaller than n of length l:

   ```
   sage: # edit here
   ```

Standard Iterables

Finally, many standard Python and Sage objects are *iterable*; that is one may iterate through their elements:

```
sage: sum( x^len(s) for s in Subsets(8) )
x^8 + 8*x^7 + 28*x^6 + 56*x^5 + 70*x^4 + 56*x^3 + 28*x^2 + 8*x + 1

sage: sum( x^p.length() for p in Permutations(3) )
x^3 + 2*x^2 + 2*x + 1

sage: factor(sum( x^p.length() for p in Permutations(3) ))
(x^2 + x + 1)*(x + 1)

sage: P = Permutations(5)
sage: all( p in P for p in P )
True

sage: for p in GL(2, 2): print p; print
[1 0]
[0 1]

[0 1]
[1 0]

[0 1]
[1 1]

[1 1]
[0 1]

[1 1]
[1 0]

[1 0]
[1 1]


sage: for p in Partitions(3): print p
```

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

Beware of infinite loops:

```
sage: for p in Partitions(): print p           # not tested
```

```
sage: for p in Primes(): print p               # not tested
```

Infinite loops can nevertheless be very useful:

```
sage: exists( Primes(), lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

```
sage: counter_examples = (p for p in Primes() if not is_prime(mersenne(p)))
sage: counter_examples.next()
11
```

6.1.10 Tutorial: Objects and Classes in Python and Sage

Author: Florent Hivert <florent.hivert@univ-rouen.fr>

This tutorial is an introduction to object-oriented programming in Python and Sage. It requires basic knowledge about imperative/procedural programming (the most common programming style) – that is, conditional instructions, loops, functions (see the "Programming" section of the Sage tutorial) – but no further knowledge about objects and classes is assumed. It is designed as an alternating sequence of formal introduction and exercises. *Solutions to the exercises* are given at the end.

Foreword: variables, names and objects

As an object-oriented language, Python's ''variables" behavior may be surprising for people used to imperative languages like C or Maple. The reason is that they are **not variables but names**.

The following explanation is borrowed from David Goodger.

Other languages have "variables"

In many other languages, assigning to a variable puts a value into a box.

```
int a = 1;
```

Box “a” now contains an integer 1.

Assigning another value to the same variable replaces the contents of the box:

```
a = 2;
```

Now box “a” contains an integer 2.

Assigning one variable to another makes a copy of the value and puts it in the new box:

```
int b = a;
```

“b” is a second box, with a copy of integer 2. Box “a” has a separate copy.

Python has “names”

In Python, a “name” or “identifier” is like a parcel tag (or nametag) attached to an object.

```
a = 1
```

Here, an integer 1 object has a tag labelled “a”.

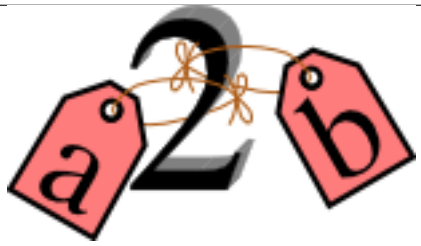
If we reassign to “a”, we just move the tag to another object:

`a = 2`		

Now the name “a” is attached to an integer 2 object.

The original integer 1 object no longer has a tag “a”. It may live on, but we can’t get to it through the name “a”. (When an object has no more references or tags, it is removed from memory.)

If we assign one name to another, we’re just attaching another nametag to an existing object:

`b = a`	

The name “b” is just a second tag bound to the same object as “a”.

Although we commonly refer to “variables” even in Python (because it’s common terminology), we really mean “names” or “identifiers”. In Python, “variables” are nametags for values, not labelled boxes.

Warning: As a consequence, when there are **two tags** “a” and “b” on the **same object**, modifying the object tagged “b” also modifies the object tagged “a”:

```
sage: a = [1,2,3]
sage: b = a
sage: b[1] = 0
sage: a
[1, 0, 3]
```

Note that reassigning the tag “b” (rather than modifying the object with that tag) doesn’t affect the object tagged “a”:

```
sage: b = 7
sage: b
7
sage: a
[1, 0, 3]
```

Object-oriented programming paradigm

The **object-oriented programming** paradigm relies on the two following fundamental rules:

1. Anything of the real (or mathematical) world which needs to be manipulated by the computer is modeled by an **object**.

2. Each object is an **instance** of some **class**.

At this point, those two rules are a little meaningless, so let's give some more or less precise definitions of the terms:

---

object a **portion of memory** which contains the information needed to model the real world thing.

class defines the **data structure** used to store the objects which are instances of the class together with their **behavior**.

Let's start with some examples: We consider the vector space over $\mathbf{Q}$ whose basis is indexed by permutations, and a particular element in it:

```
sage: F = CombinatorialFreeModule(QQ, Permutations())
sage: el = 3*F([1,3,2])+ F([1,2,3])
sage: el
B[[1, 2, 3]] + 3*B[[1, 3, 2]]
```

(For each permutation, say `[1, 3, 2]`, the corresponding element in `F` is denoted by `B[[1, 3, 2]]` – in a `CombinatorialFreeModule`, if an element is indexed by `x`, then by default its print representation is `B[x]`.)

In Python, everything is an object so there isn't any difference between types and classes. One can get the class of the object `el` by:

```
sage: type(el)
<class 'sage.combinat.free_module.CombinatorialFreeModule_with_category.element_class'>
```

As such, this is not very informative. We'll come back to it later. The data associated to objects are stored in so-called **attributes**. They are accessed through the syntax `obj.attribute_name`. For an element of a combinatorial free module, the main attribute is called `_monomial_coefficients`. It is a dictionary associating coefficients to indices:

```
sage: el._monomial_coefficients
{[1, 2, 3]: 1, [1, 3, 2]: 3}
```

Modifying the attribute modifies the objects:

```
sage: el._monomial_coefficients[Permutation([3,2,1])] = 1/2
sage: el
B[[1, 2, 3]] + 3*B[[1, 3, 2]] + 1/2*B[[3, 2, 1]]
```

Warning: as a user, you are *not* supposed to do such a modification by yourself (see note on *private attributes* below).

As an element of a vector space, `el` has a particular behavior:

```
sage: 2*el
2*B[[1, 2, 3]] + 6*B[[1, 3, 2]] + B[[3, 2, 1]]
sage: el.support()
[[1, 2, 3], [1, 3, 2], [3, 2, 1]]
sage: el.coefficient([1, 2, 3])
1
```

The behavior is defined through **methods** (`support`, `coefficient`). Note that this is true even for equality, printing or mathematical operations. For example, the call `a == b` actually is translated to the method call `a.__eq__(b)`. The names of those special methods which are usually called through operators are fixed by the Python language and are of the form `__name__`. Examples include `__eq__` and `__le__` for operators == and <=,

__repr__ (see *Sage specifics about classes*) for printing, __add__ and __mult__ for operators + and *. See http://docs.python.org/library/ for a complete list.

```
sage: el.__eq__(F([1,3,2]))
False
sage: el.__repr__()
'B[[1, 2, 3]] + 3*B[[1, 3, 2]] + 1/2*B[[3, 2, 1]]'
sage: el.__mul__(2)
2*B[[1, 2, 3]] + 6*B[[1, 3, 2]] + B[[3, 2, 1]]
```

Some particular actions modify the data structure of el:

```
sage: el.rename("bla")
sage: el
bla
```

Note: The class is stored in a particular attribute called __class__, and the normal attributes are stored in a dictionary called __dict__:

```
sage: F = CombinatorialFreeModule(QQ, Permutations())
sage: el = 3*F([1,3,2])+ F([1,2,3])
sage: el.rename("foo")
sage: el.__class__
 <class 'sage.combinat.free_module.CombinatorialFreeModule_with_category.element_class'>
sage: el.__dict__
{'_monomial_coefficients': {[1, 2, 3]: 1, [1, 3, 2]: 3}, '__custom_name': 'foo'}
```

Lots of Sage objects are not Python objects but compiled Cython objects. Python sees them as builtin objects and you don't have access to the data structure. Examples include integers and permutation group elements:

```
sage: e = Integer(9)
sage: type(e)
<type 'sage.rings.integer.Integer'>
sage: e.__dict__
dict_proxy({'__module__': 'sage.categories.euclidean_domains',
'euclidean_degree': <abstract method euclidean_degree at 0x...>,
'_reduction': (<built-in function getattr>, (Category of euclidean
domains, 'element_class')), 'gcd':
<sage.structure.element.NamedBinopMethod object at 0x...>,
'_sage_src_lines_': <staticmethod object at 0x...>, 'quo_rem':
<abstract method quo_rem at 0x...>, '__doc__': None})
sage: e.__dict__.keys()
['__module__', 'euclidean_degree', '_reduction', 'gcd', '_sage_src_lines_', 'quo_rem', '__doc__']

sage: id4 = SymmetricGroup(4).one()
sage: type(id4)
<type 'sage.groups.perm_gps.permgroup_element.PermutationGroupElement'>
sage: id4.__dict__
dict_proxy({'__module__': 'sage.categories.category',
'_reduction': (<built-in function getattr>,
               (Join of Category of finite permutation groups
                    and Category of finite weyl groups, 'element_class')),
'__doc__': "...",
'_sage_src_lines_': <staticmethod object at 0x...>})
```

Note: Each object corresponds to a portion of memory called its **identity** in Python. You can get the identity using `id`:

```
sage: el = Integer(9)
sage: id(el)  # random
139813642977744
sage: el1 = el; id(el1) == id(el)
True
sage: el1 is el
True
```

In Python (and therefore in Sage), two objects with the same identity will be equal, but the converse is not true in general. Thus the identity function is different from mathematical identity:

```
sage: el2 = Integer(9)
sage: el2 == el1
True
sage: el2 is el1
False
sage: id(el2) == id(el)
False
```

Summary

To define some object, you first have to write a **class**. The class will define the methods and the attributes of the object.

method particular kind of function associated with an object used to get information about the object or to manipulate it.

attribute variable where information about the object is stored.

An example: glass of beverage in a restaurant

Let's write a small class about glasses in a restaurant:

```
sage: class Glass(object):
...       def __init__(self, size):
...           assert size > 0
...           self._size = float(size)  # an attribute
...           self._content = float(0.0)  # another attribute
...       def __repr__(self):
...           if self._content == 0.0:
...               return "An empty glass of size %s"%(self._size)
...           else:
...               return "A glass of size %s cl containing %s cl of water"%(
...                       self._size, self._content)
...       def fill(self):
...           self._content = self._size
...       def empty(self):
...           self._content = float(0.0)
```

Let's create a small glass:

```
sage: myGlass = Glass(10); myGlass
An empty glass of size 10.0
```

```
sage: myGlass.fill(); myGlass
A glass of size 10.0 cl containing 10.0 cl of water
sage: myGlass.empty(); myGlass
An empty glass of size 10.0
```

Some comments:

1. The definition of the class `Glass` defines two attributes, `_size` and `_content`. It defines four methods, `__init__`, `__repr__`, `fill`, and `empty`. (Any instance of this class will also have other attributes and methods, inherited from the class `object`. See *Inheritance* below.)
2. The method `__init__` is used to initialize the object: it is used by the so-called **constructor** of the class that is executed when calling `Glass(10)`.
3. The method `__repr__` returns a string which is used to print the object, for example in this case when evaluating `myGlass`.

Note: Private Attributes

- Most of the time, in order to ensure consistency of the data structures, the user is not supposed to directly change certain attributes of an object. Those attributes are called **private**. Since there is no mechanism to ensure privacy in Python, the convention is the following: private attributes have names beginning with an underscore.
- As a consequence, attribute access is only made through methods. Methods for reading or writing a private attribute are called accessors.
- Methods which are only for internal use are also prefixed with an underscore.

Exercises

1. Add a method `is_empty` which returns true if a glass is empty.
2. Define a method `drink` with a parameter `amount` which allows one to partially drink the water in the glass. Raise an error if one asks to drink more water than there is in the glass or a negative amount of water.
3. Allows the glass to be filled with wine, beer or another beverage. The method `fill` should accept a parameter `beverage`. The beverage is stored in an attribute `_beverage`. Update the method `__repr__` accordingly.
4. Add an attribute `_clean` and methods `is_clean` and `wash`. At the creation a glass is clean, as soon as it's filled it becomes dirty, and it must be washed to become clean again.
5. Test everything.
6. Make sure that everything is tested.
7. Test everything again.

Inheritance

The problem: objects of **different** classes may share a **common behavior**.

For example, if one wants to deal with different dishes (forks, spoons, ...), then there is common behavior (becoming dirty and being washed). So the different classes associated to the different kinds of dishes should have the same `clean`, `is_clean` and `wash` methods. But copying and pasting code is very bad for maintenance: mistakes are copied, and to change anything one has to remember the location of all the copies. So there is a need for a mechanism which allows the programmer to factorize the common behavior. It is called **inheritance** or **sub-classing**: one writes a base class which factorizes the common behavior and then reuses the methods from this class.

We first write a small class ''AbstractDish" which implements the "clean-dirty-wash" behavior:

```
sage: class AbstractDish(object):
...       def __init__(self):
...           self._clean = True
...       def is_clean(self):
...           return self._clean
...       def state(self):
...           return "clean" if self.is_clean() else "dirty"
...       def __repr__(self):
...           return "An unspecified %s dish"%self.state()
...       def _make_dirty(self):
...           self._clean = False
...       def wash(self):
...           self._clean = True
```

Now one can reuse this behavior within a class `Spoon`:

```
sage: class Spoon(AbstractDish):  # Spoon inherits from AbstractDish
...       def __repr__(self):
...           return "A %s spoon"%self.state()
...       def eat_with(self):
...           self._make_dirty()
```

Let's test it:

```
sage: s = Spoon(); s
A clean spoon
sage: s.is_clean()
True
sage: s.eat_with(); s
A dirty spoon
sage: s.is_clean()
False
sage: s.wash(); s
A clean spoon
```

Summary

1. Any class can reuse the behavior of another class. One says that the subclass **inherits** from the superclass or that it **derives** from it.
2. Any instance of the subclass is also an instance of its superclass:

   ```
   sage: type(s)
   <class '__main__.Spoon'>
   sage: isinstance(s, Spoon)
   True
   sage: isinstance(s, AbstractDish)
   True
   ```

3. If a subclass redefines a method, then it replaces the former one. One says that the subclass **overloads** the method. One can nevertheless explicitly call the hidden superclass method.

   ```
   sage: s.__repr__()
   'A clean spoon'
   sage: Spoon.__repr__(s)
   'A clean spoon'
   ```

```
sage: AbstractDish.__repr__(s)
'An unspecified clean dish'
```

Note: Advanced superclass method call

Sometimes one wants to call an overloaded method without knowing in which class it is defined. To do this, use the `super` operator:

```
sage: super(Spoon, s).__repr__()
'An unspecified clean dish'
```

A very common usage of this construct is to call the `__init__` method of the superclass:

```
sage: class Spoon(AbstractDish):
...       def __init__(self):
...           print "Building a spoon"
...           super(Spoon, self).__init__()
...       def __repr__(self):
...           return "A %s spoon"%self.state()
...       def eat_with(self):
...           self._make_dirty()
sage: s = Spoon()
Building a spoon
sage: s
A clean spoon
```

Exercises

1. Modify the class `Glasses` so that it inherits from `Dish`.
2. Write a class `Plate` whose instance can contain any meal together with a class `Fork`. Avoid as much as possible code duplication (hint: you can write a factorized class `ContainerDish`).
3. Test everything.

Sage specifics about classes

Compared to Python, Sage has particular ways to handle objects:

- Any classes for mathematical objects in Sage should inherit from `SageObject` rather than from `object`. Most of the time, they actually inherit from a subclass such as `Parent` or `Element`.
- Printing should be done through `_repr_` instead of `__repr__` to allow for renaming.
- More generally, Sage-specific special methods are usually named `_meth_` rather than `__meth__`. For example, lots of classes implement `_hash_` which is used and cached by `__hash__`. In the same vein, elements of a group usually implement `_mul_`, so that there is no need to take care about coercions as they are done in `__mul__`.

For more details, see the Sage Developer's Guide.

Solutions to the exercises

1. Here is a solution to the first exercise:

```
sage: class Glass(object):
...       def __init__(self, size):
...           assert size > 0
...           self._size = float(size)
...           self.wash()
...       def __repr__(self):
...           if self._content == 0.0:
...               return "An empty glass of size %s"%(self._size)
...           else:
...               return "A glass of size %s cl containing %s cl of %s"%(
...                       self._size, self._content, self._beverage)
...       def content(self):
...           return self._content
...       def beverage(self):
...           return self._beverage
...       def fill(self, beverage = "water"):
...           if not self.is_clean():
...               raise ValueError("Don't want to fill a dirty glass")
...           self._clean = False
...           self._content = self._size
...           self._beverage = beverage
...       def empty(self):
...           self._content = float(0.0)
...       def is_empty(self):
...           return self._content == 0.0
...       def drink(self, amount):
...           if amount <= 0.0:
...               raise ValueError("amount must be positive")
...           elif amount > self._content:
...               raise ValueError("not enough beverage in the glass")
...           else:
...               self._content -= float(amount)
...       def is_clean(self):
...           return self._clean
...       def wash(self):
...           self._content = float(0.0)
...           self._beverage = None
...           self._clean = True
```

2. Let's check that everything is working as expected:

```
sage: G = Glass(10.0)
sage: G
An empty glass of size 10.0
sage: G.is_empty()
True
sage: G.drink(2)
Traceback (most recent call last):
...
ValueError: not enough beverage in the glass
sage: G.fill("beer")
sage: G
A glass of size 10.0 cl containing 10.0 cl of beer
sage: G.is_empty()
False
sage: G.is_clean()
False
sage: G.drink(5.0)
```

```
sage: G
A glass of size 10.0 cl containing 5.0 cl of beer
sage: G.is_empty()
False
sage: G.is_clean()
False
sage: G.drink(5)
sage: G
An empty glass of size 10.0
sage: G.is_clean()
False
sage: G.fill("orange juice")
Traceback (most recent call last):
...
ValueError: Don't want to fill a dirty glass
sage: G.wash()
sage: G
An empty glass of size 10.0
sage: G.fill("orange juice")
sage: G
A glass of size 10.0 cl containing 10.0 cl of orange juice
```

3. Here is the solution to the second exercice:

```
sage: class AbstractDish(object):
...       def __init__(self):
...           self._clean = True
...       def is_clean(self):
...           return self._clean
...       def state(self):
...           return "clean" if self.is_clean() else "dirty"
...       def __repr__(self):
...           return "An unspecified %s dish"%self.state()
...       def _make_dirty(self):
...           self._clean = False
...       def wash(self):
...           self._clean = True


sage: class ContainerDish(AbstractDish):
...       def __init__(self, size):
...           assert size > 0
...           self._size = float(size)
...           self._content = float(0)
...           super(ContainerDish, self).__init__()
...       def content(self):
...           return self._content
...       def empty(self):
...           self._content = float(0.0)
...       def is_empty(self):
...           return self._content == 0.0
...       def wash(self):
...           self._content = float(0.0)
...           super(ContainerDish, self).wash()


sage: class Glass(ContainerDish):
...       def __repr__(self):
```

```
...             if self._content == 0.0:
...                 return "An empty glass of size %s"%(self._size)
...             else:
...                 return "A glass of size %s cl containing %s cl of %s"%(
...                         self._size, self._content, self._beverage)
...         def beverage(self):
...             return self._beverage
...         def fill(self, beverage = "water"):
...             if not self.is_clean():
...                 raise ValueError("Don't want to fill a dirty glass")
...             self._make_dirty()
...             self._content = self._size
...             self._beverage = beverage
...         def drink(self, amount):
...             if amount <= 0.0:
...                 raise ValueError("amount must be positive")
...             elif amount > self._content:
...                 raise ValueError("not enough beverage in the glass")
...             else:
...                 self._content -= float(amount)
...         def wash(self):
...             self._beverage = None
...             super(Glass, self).wash()
```

4. Let's check that everything is working as expected:

```
sage: G = Glass(10.0)
sage: G
An empty glass of size 10.0
sage: G.is_empty()
True
sage: G.drink(2)
Traceback (most recent call last):
...
ValueError: not enough beverage in the glass
sage: G.fill("beer")
sage: G
A glass of size 10.0 cl containing 10.0 cl of beer
sage: G.is_empty()
False
sage: G.is_clean()
False
sage: G.drink(5.0)
sage: G
A glass of size 10.0 cl containing 5.0 cl of beer
sage: G.is_empty()
False
sage: G.is_clean()
False
sage: G.drink(5)
sage: G
An empty glass of size 10.0
sage: G.is_clean()
False
sage: G.fill("orange juice")
Traceback (most recent call last):
...
ValueError: Don't want to fill a dirty glass
```

```
sage: G.wash()
sage: G
An empty glass of size 10.0
sage: G.fill("orange juice")
sage: G
A glass of size 10.0 cl containing 10.0 cl of orange juice
```

Todo

give the example of the class `Plate`.

That all folks !

6.1.11 Functional Programming for Mathematicians

Author: Minh Van Nguyen <nguyenminh2@gmail.com>

This tutorial discusses some techniques of functional programming that might be of interest to mathematicians or people who use Python for scientific computation. We start off with a brief overview of procedural and object-oriented programming, and then discuss functional programming techniques. Along the way, we briefly review Python's built-in support for functional programming, including filter, lambda, map and reduce. The tutorial concludes with some resources on detailed information on functional programming using Python.

Styles of programming

Python supports several styles of programming. You could program in the procedural style by writing a program as a list of instructions. Say you want to implement addition and multiplication over the integers. A procedural program to do so would be as follows:

```
sage: def add_ZZ(a, b):
...       return a + b
...
sage: def mult_ZZ(a, b):
...       return a * b
...
sage: add_ZZ(2, 3)
5
sage: mult_ZZ(2, 3)
6
```

The Python module operator defines several common arithmetic and comparison operators as named functions. Addition is defined in the built-in function `operator.add` and multiplication is defined in `operator.mul`. The above example can be worked through as follows:

```
sage: from operator import add
sage: from operator import mul
sage: add(2, 3)
5
sage: mul(2, 3)
6
```

Another common style of programming is called object-oriented programming. Think of an object as code that encapsulates both data and functionalities. You could encapsulate integer addition and multiplication as in the following object-oriented implementation:

```
sage: class MyInteger:
...       def __init__(self):
...           self.cardinality = "infinite"
...       def add(self, a, b):
...           return a + b
...       def mult(self, a, b):
...           return a * b
...
sage: myZZ = MyInteger()
sage: myZZ.cardinality
'infinite'
sage: myZZ.add(2, 3)
5
sage: myZZ.mult(2, 3)
6
```

Functional programming using map

Functional programming is yet another style of programming in which a program is decomposed into various functions. The Python built-in functions `map`, `reduce` and `filter` allow you to program in the functional style. The function

```
map(func, seq1, seq2, ...)
```

takes a function `func` and one or more sequences, and apply `func` to elements of those sequences. In particular, you end up with a list like so:

```
[func(seq1[0], seq2[0], ...), func(seq1[1], seq2[1], ...), ...]
```

In many cases, using `map` allows you to express the logic of your program in a concise manner without using list comprehension. For example, say you have two lists of integers and you want to add them element-wise. A list comprehension to accomplish this would be as follows:

```
sage: A = [1, 2, 3, 4]
sage: B = [2, 3, 5, 7]
sage: [A[i] + B[i] for i in range(len(A))]
[3, 5, 8, 11]
```

Alternatively, you could use the Python built-in addition function `operator.add` together with `map` to achieve the same result:

```
sage: from operator import add
sage: A = [1, 2, 3, 4]
sage: B = [2, 3, 5, 7]
sage: map(add, A, B)
[3, 5, 8, 11]
```

An advantage of `map` is that you do not need to explicitly define a for loop as was done in the above list comprehension.

Define small functions using lambda

There are times when you want to write a short, one-liner function. You could re-write the above addition function as follows:

```
sage: def add_ZZ(a, b): return a + b
...
```

Or you could use a `lambda` statement to do the same thing in a much clearer style. The above addition and multiplication functions could be written using `lambda` as follows:

```
sage: add_ZZ = lambda a, b: a + b
sage: mult_ZZ = lambda a, b: a * b
sage: add_ZZ(2, 3)
5
sage: mult_ZZ(2, 3)
6
```

Things get more interesting once you combine `map` with the `lambda` statement. As an exercise, you might try to write a simple function that implements a constructive algorithm for the Chinese Remainder Theorem. You could use list comprehension together with `map` and `lambda` as shown below. Here, the parameter `A` is a list of integers and `M` is a list of moduli.

```
sage: def crt(A, M):
...       Mprod = prod(M)
...       Mdiv = map(lambda x: Integer(Mprod / x), M)
...       X = map(inverse_mod, Mdiv, M)
...       x = sum([A[i]*X[i]*Mdiv[i] for i in range(len(A))])
...       return mod(x, Mprod).lift()
...
sage: A = [2, 3, 1]
sage: M = [3, 4, 5]
sage: x = crt(A, M); x
11
sage: mod(x, 3)
2
sage: mod(x, 4)
3
sage: mod(x, 5)
1
```

To produce a random matrix over a ring, say $\mathbf{Z}$, you could start by defining a matrix space and then obtain a random element of that matrix space:

```
sage: MS = MatrixSpace(ZZ, nrows=5, ncols=3)
sage: MS.random_element()  # random

[ 6  1  0]
[-1  5  0]
[-1  0  0]
[-5  0  1]
[ 1 -1 -3]
```

Or you could use the function `random_matrix`:

```
sage: random_matrix(ZZ, nrows=5, ncols=3)  # random

[  2 -50   0]
[ -1   0  -6]
[ -4  -1  -1]
[  1   1   3]
[  2  -1  -1]
```

The next example uses `map` to construct a list of random integer matrices:

```
sage: rows = [randint(1, 10) for i in range(10)]
sage: cols = [randint(1, 10) for i in range(10)]
sage: rings = [ZZ]*10
sage: M = map(random_matrix, rings, rows, cols)
sage: M[0]  # random

[ -1  -3  -1 -37   1  -1  -4   5]
[  2   1   1   5   2   1  -2   1]
[ -1   0  -4   0  -2   1  -2   1]
```

If you want more control over the entries of your matrices than the `random_matrix` function permits, you could use `lambda` together with `map` as follows:

```
sage: rand_row = lambda n: [randint(1, 10) for i in range(n)]
sage: rand_mat = lambda nrows, ncols: [rand_row(ncols) for i in range(nrows)]
sage: matrix(rand_mat(5, 3))  # random

[ 2  9 10]
[ 8  8  9]
[ 6  7  6]
[ 9  2 10]
[ 2  6  2]
sage: rows = [randint(1, 10) for i in range(10)]
sage: cols = [randint(1, 10) for i in range(10)]
sage: M = map(rand_mat, rows, cols)
sage: M = map(matrix, M)
sage: M[0]  # random

[ 9  1  5  2 10 10  1]
[ 3  4  3  7  4  3  7]
[ 4  8  7  6  4  2 10]
[ 1  6  3  3  6  2  1]
[ 5  5  2  6  4  3  4]
[ 6  6  2  9  4  5  1]
[10  2  5  5  7 10  4]
[ 2  7  3  5 10  8  1]
[ 1  5  1  7  8  8  6]
```

Reducing a sequence to a value

The function `reduce` takes a function of two arguments and apply it to a given sequence to reduce that sequence to a single value. The function sum is an example of a `reduce` function. The following sample code uses `reduce` and the built-in function `operator.add` to add together all integers in a given list. This is followed by using `sum` to accomplish the same task:

```
sage: from operator import add
sage: L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: reduce(add, L)
55
sage: sum(L)
55
```

In the following sample code, we consider a vector as a list of real numbers. The dot product is then implemented using the functions `operator.add` and `operator.mul`, in conjunction with the built-in Python functions `reduce` and `map`. We then show how `sum` and `map` could be combined to produce the same result.

```
sage: from operator import add
sage: from operator import mul
sage: U = [1, 2, 3]
sage: V = [2, 3, 5]
sage: reduce(add, map(mul, U, V))
23
sage: sum(map(mul, U, V))
23
```

Or you could use Sage's built-in support for the dot product:

```
sage: u = vector([1, 2, 3])
sage: v = vector([2, 3, 5])
sage: u.dot_product(v)
23
```

Here is an implementation of the Chinese Remainder Theorem without using `sum` as was done previously. The version below uses `operator.add` and defines `mul3` to multiply three numbers instead of two.

```
sage: def crt(A, M):
...       from operator import add
...       Mprod = prod(M)
...       Mdiv = map(lambda x: Integer(Mprod / x), M)
...       X = map(inverse_mod, Mdiv, M)
...       mul3 = lambda a, b, c: a * b * c
...       x = reduce(add, map(mul3, A, X, Mdiv))
...       return mod(x, Mprod).lift()
...
sage: A = [2, 3, 1]
sage: M = [3, 4, 5]
sage: x = crt(A, M); x
11
```

Filtering with filter

The Python built-in function `filter` takes a function of one argument and a sequence. It then returns a list of all those items from the given sequence such that any item in the new list results in the given function returning `True`. In a sense, you are filtering out all items that satisfy some condition(s) defined in the given function. For example, you could use `filter` to filter out all primes between 1 and 50, inclusive.

```
sage: filter(is_prime, [1..50])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

For a given positive integer n, the Euler phi function counts the number of integers a, with $1 \leq a \leq n$, such that $\gcd(a, n) = 1$. You could use list comprehension to obtain all such a's when $n = 20$:

```
sage: [k for k in range(1, 21) if gcd(k, 20) == 1]
[1, 3, 7, 9, 11, 13, 17, 19]
```

A functional approach is to use `lambda` to define a function that determines whether or not a given integer is relatively prime to 20. Then you could use `filter` instead of list comprehension to obtain all the required a's.

```
sage: is_coprime = lambda k: gcd(k, 20) == 1
sage: filter(is_coprime, range(1, 21))
[1, 3, 7, 9, 11, 13, 17, 19]
```

The function `primroots` defined below returns all primitive roots modulo a given positive prime integer p. It uses `filter` to obtain a list of integers between 1 and $p-1$, inclusive, each integer in the list being relatively prime to the order of the multiplicative group $(\mathbf{Z}/p\mathbf{Z})^*$.

```
sage: def primroots(p):
...       g = primitive_root(p)
...       znorder = p - 1
...       is_coprime = lambda x: gcd(x, znorder) == 1
...       good_odd_integers = filter(is_coprime, [1..p-1, step=2])
...       all_primroots = [power_mod(g, k, p) for k in good_odd_integers]
...       all_primroots.sort()
...       return all_primroots
...
sage: primroots(3)
[2]
sage: primroots(5)
[2, 3]
sage: primroots(7)
[3, 5]
sage: primroots(11)
[2, 6, 7, 8]
sage: primroots(13)
[2, 6, 7, 11]
sage: primroots(17)
[3, 5, 6, 7, 10, 11, 12, 14]
sage: primroots(23)
[5, 7, 10, 11, 14, 15, 17, 19, 20, 21]
sage: primroots(29)
[2, 3, 8, 10, 11, 14, 15, 18, 19, 21, 26, 27]
sage: primroots(31)
[3, 11, 12, 13, 17, 21, 22, 24]
```

Further resources

This has been a rather short tutorial to functional programming with Python. The Python standard documentation has a list of built-in functions, many of which are useful in functional programming. For example, you might want to read up on all, any, max, min, and zip. The Python module operator has numerous built-in arithmetic and comparison operators, each operator being implemented as a function whose name reflects its intended purpose. For arithmetic and comparison operations, it is recommended that you consult the `operator` module to determine if there is a built-in function that satisfies your requirement. The module itertools has numerous built-in functions to efficiently process sequences of items. The functions `filter`, `map` and `zip` have their counterparts in `itertools` as itertools.ifilter, itertools.imap and itertools.izip.

Another useful resource for functional programming in Python is the Functional Programming HOWTO by A. M. Kuchling. Steven F. Lott's book Building Skills in Python has a chapter on Functional Programming using Collections. See also the chapter Functional Programming from Mark Pilgrim's book Dive Into Python.

You might also want to consider experimenting with Haskell for expressing mathematical concepts. For an example of Haskell in expressing mathematical algorithms, see J. Gibbons' article Unbounded Spigot Algorithms for the Digits of Pi in the American Mathematical Monthly.

6.1.12 How to implement new algebraic structures in Sage

Sage's category and coercion framework

Author: Simon King, Friedrich–Schiller–Universität Jena, <simon.king@uni-jena.de>

The aim of this tutorial is to explain how one can benefit from Sage's category framework and coercion model when implementing new algebraic structures. It is based on a worksheet created in 2011.

We illustrate the concepts of Sage's category framework and coercion model by means of a detailed example, namely a toy implementation of fraction fields. The code is developed step by step, so that the reader can focus on one detail in each part of this tutorial. The complete code can be found in the appendix.

Outline

- Use existing base classes

  For using Sage's coercion system, it is essential to work with sub–classes of `sage.structure.parent.Parent` or `sage.structure.element.Element`, respectively. They provide default implementations of many "magical" double-underscore Python methods, which must not be overridden. Instead, the actual implementation should be in *single underscore* methods, such as `_add_` or `_mul_`.

- Turn your parent structure into an object of a category

  Declare the category during initialisation—Your parent structure will inherit further useful methods and consistency tests.

- Provide your parent structure with an element class

  Assign to it an attribute called `Element`—The elements will inherit further useful methods from the category. In addition, some basic conversions will immediately work.

- Implement further conversions

  Never override a parent's `__call__` method! Provide the method `_element_constructor_` instead.

- Declare coercions

  If a conversion happens to be a morphism, you may consider to turn it into a coercion. It will then *implicitly* be used in arithmetic operations.

- Advanced coercion: Define construction functors for your parent structure

  Sage will automatically create new parents for you when needed, by the so–called `sage.categories.pushout.pushout()` construction.

- Run the automatic test suites

  Each method should be documented and provide a doc test (we are not giving examples here). In addition, any method defined for the objects or elements of a category should be supported by a test method, that is executed when running the test suite.

Base classes

In Sage, a "Parent" is an object of a category and contains elements. Parents should inherit from `sage.structure.parent.Parent` and their elements from `sage.structure.element.Element`.

Sage provides appropriate sub–classes of `Parent` and `Element` for a variety of more concrete algebraic structures, such as groups, rings, or fields, and of their elements. But some old stuff in Sage doesn't use it. **Volunteers for refactoring are welcome!**

The parent Since we wish to implement a special kind of fields, namely fraction fields, it makes sense to build on top of the base class `sage.rings.ring.Field` provided by Sage.

```
sage: from sage.rings.ring import Field
```

This base class provides a lot more methods than a general parent:

```
sage: [p for p in dir(Field) if p not in dir(Parent)]
['__div__', '__fraction_field', '__ideal_monoid', '__iter__',
 '__pow__', '__rdiv__', '__rpow__', '__rxor__', '__xor__',
 '_an_element', '_an_element_c', '_an_element_impl', '_coerce_',
 '_coerce_c', '_coerce_impl', '_coerce_self', '_coerce_try',
 '_default_category', '_gens', '_gens_dict',
 '_has_coerce_map_from', '_ideal_class_', '_latex_names', '_list',
 '_one_element', '_pseudo_fraction_field',
 '_random_nonzero_element', '_richcmp', '_unit_ideal',
 '_zero_element', '_zero_ideal', 'algebraic_closure',
 'base_extend', 'cardinality', 'class_group', 'coerce_map_from_c',
 'coerce_map_from_impl', 'content', 'divides', 'extension',
 'fraction_field', 'frobenius_endomorphism', 'gcd', 'gen', 'gens',
 'get_action_c', 'get_action_impl', 'has_coerce_map_from_c',
 'has_coerce_map_from_impl', 'ideal', 'ideal_monoid',
 'integral_closure', 'is_commutative', 'is_field', 'is_finite',
 'is_integral_domain', 'is_integrally_closed', 'is_noetherian',
 'is_prime_field', 'is_ring', 'is_subring',
 'krull_dimension', 'list', 'ngens', 'one', 'one_element',
 'order', 'prime_subfield', 'principal_ideal', 'quo', 'quotient',
 'quotient_ring', 'random_element', 'unit_ideal', 'zero',
 'zero_element', 'zero_ideal', 'zeta', 'zeta_order']
```

The following is a very basic implementation of fraction fields, that needs to be complemented later.

```
sage: from sage.structure.unique_representation import UniqueRepresentation
sage: class MyFrac(UniqueRepresentation, Field):
....:     def __init__(self, base):
....:         if base not in IntegralDomains():
....:             raise ValueError, "%s is no integral domain"%base
....:         Field.__init__(self, base)
....:     def _repr_(self):
....:         return "NewFrac(%s)"%repr(self.base())
....:     def base_ring(self):
....:         return self.base().base_ring()
....:     def characteristic(self):
....:         return self.base().characteristic()
```

This basic implementation is formed by the following steps:

- Any ring in Sage has a **base** and a **base ring**. The "usual" fraction field of a ring R has the base R and the base ring `R.base_ring()`:

  ```
  sage: Frac(QQ['x']).base(), Frac(QQ['x']).base_ring()
  (Univariate Polynomial Ring in x over Rational Field, Rational Field)
  ```

  Declaring the base is easy: We just pass it as an argument to the field constructor.

```
sage: Field(ZZ['x']).base()
Univariate Polynomial Ring in x over Integer Ring
```

We are implementing a seperate method returning the base ring.

- Python uses double–underscore methods for arithemetic methods and string representations. Sage's base classes often have a default implementation, and it is requested to **implement SINGLE underscore methods _repr_, and similarly _add_, _mul_ etc.**
- You are encouraged to **make your parent "unique"**. That's to say, parents should only evaluate equal if they are identical. Sage provides frameworks to create unique parents. We use here the most easy one: Inheriting from the class `sage.structure.unique_representation.UniqueRepresentation` is enough. Making parents unique can be quite important for an efficient implementation, because the repeated creation of "the same" parent would take a lot of time.
- Fraction fields are only defined for integral domains. Hence, we raise an error if the given ring does not belong to the category of integral domains. This is our first use case of categories.
- Last, we add a method that returns the characteristic of the field. We don't go into details, but some automated tests that we study below implicitly rely on this method.

We see that our basic implementation correctly refuses a ring that is not an integral domain:

```
sage: MyFrac(ZZ['x'])
NewFrac(Univariate Polynomial Ring in x over Integer Ring)
sage: MyFrac(Integers(15))
Traceback (most recent call last):
...
ValueError: Ring of integers modulo 15 is no integral domain
```

Note: Inheritance from `UniqueRepresentation` automatically provides our class with pickling, preserving the unique parent condition. If we had defined the class in some external module or in an interactive session, pickling would work immediately.

However, for making the following example work in Sage's doctesting framework, we need to assign our class as an attribute of the `__main__` module, so that the class can be looked up during unpickling.

```
sage: import __main__
sage: __main__.MyFrac = MyFrac
sage: loads(dumps(MyFrac(ZZ))) is MyFrac(ZZ)
True
```

Note: In the following sections, we will successively add or change details of `MyFrac`. Rather than giving a full class definition in each step, we define new versions of `MyFrac` by inheriting from the previously defined version of `MyFrac`. We believe this will help the reader to focus on the single detail that is relevant in each section.

The complete code can be found in the appendix.

The elements We use the base class `sage.structure.element.FieldElement`. Note that in the creation of field elements it is not tested that the given parent is a field:

```
sage: from sage.structure.element import FieldElement
sage: FieldElement(ZZ)
Generic element of a structure
```

Our toy implementation of fraction field elements is based on the following considerations:

- A fraction field element is defined by numerator and denominator, which both need to be elements of the base. There should be methods returning numerator resp. denominator.
- The denominator must not be zero, and (provided that the base is an ordered ring) we can make it non-negative, without loss of generality. By default, the denominator is one.
- The string representation is returned by the single–underscore method `_repr_`. In order to make our fraction field elements distinguishable from those already present in Sage, we use a different string representation.
- Arithmetic is implemented in single–underscore method `_add_`, `_mul_`, etc. **We do not override the default double underscore __add__, __mul__**, since otherwise, we could not use Sage's coercion model.
- In the single underscore methods and in `__cmp__`, we can assume that *both arguments belong to the same parent*. This is one benefit of the coercion model. Note that `__cmp__` should be provided, since otherwise comparison does not work in the way expected in Python:

```
sage: class Foo(sage.structure.element.Element):
....:  def __init__(self, parent, x):
....:      self.x = x
....:  def _repr_(self):
....:      return "<%s>"%self.x
sage: a = Foo(ZZ, 1)
sage: b = Foo(ZZ, 2)
sage: cmp(a,b)
Traceback (most recent call last):
...
NotImplementedError: BUG: sort algorithm for elements of 'None' not implemented
```

- When constructing new elements as the result of arithmetic operations, we do not directly name our class, but we use `self.__class__`. Later, this will come in handy.

This gives rise to the following code:

```
sage: class MyElement(FieldElement):
....:     def __init__(self, parent,n,d=None):
....:         B = parent.base()
....:         if d is None:
....:             d = B.one_element()
....:         if n not in B or d not in B:
....:             raise ValueError("Numerator and denominator must be elements of %s"%B)
....:         # Numerator and denominator should not just be "in" B,
....:         # but should be defined as elements of B
....:         d = B(d)
....:         n = B(n)
....:         if d==0:
....:             raise ZeroDivisionError("The denominator must not be zero")
....:         if d<0:
....:             self.n = -n
....:             self.d = -d
....:         else:
....:             self.n = n
....:             self.d = d
....:         FieldElement.__init__(self,parent)
....:     def numerator(self):
....:         return self.n
....:     def denominator(self):
....:         return self.d
....:     def _repr_(self):
....:         return "(%s):(%s)"%(self.n,self.d)
```

```
....:     def __cmp__(self, other):
....:         return cmp(self.n*other.denominator(), other.numerator()*self.d)
....:     def _add_(self, other):
....:         C = self.__class__
....:         D = self.d*other.denominator()
....:         return C(self.parent(), self.n*other.denominator()+self.d*other.numerator(), D)
....:     def _sub_(self, other):
....:         C = self.__class__
....:         D = self.d*other.denominator()
....:         return C(self.parent(), self.n*other.denominator()-self.d*other.numerator(),D)
....:     def _mul_(self, other):
....:         C = self.__class__
....:         return C(self.parent(), self.n*other.numerator(), self.d*other.denominator())
....:     def _div_(self, other):
....:         C = self.__class__
....:         return C(self.parent(), self.n*other.denominator(), self.d*other.numerator())
```

Features and limitations of the basic implementation Thanks to the single underscore methods, some basic arithmetics works, **if** we stay inside a single parent structure:

```
sage: P = MyFrac(ZZ)
sage: a = MyElement(P, 3, 4)
sage: b = MyElement(P, 1, 2)
sage: a+b, a-b, a*b, a/b
((10):(8), (2):(8), (3):(8), (6):(4))
sage: a-b == MyElement(P, 1, 4)
True
```

We didn't implement exponentiation—but it just works:

```
sage: a^3
(27):(64)
```

There is a default implementation of element tests. We can already do

```
sage: a in P
True
```

since a is defined as an element of P. However, we can not verify yet that the integers are contained in the fraction field of the ring of integers. It does not even give a wrong answer, but results in an error:

```
sage: 1 in P
Traceback (most recent call last):
...
NotImplementedError
```

We will take care of this later.

Categories in Sage

Sometimes the base classes do not reflect the mathematics: The set of $m \times n$ matrices over a field forms, in general, not more than a vector space. Hence, this set (called `MatrixSpace`) is not implemented on top of `sage.rings.ring.Ring`. However, if $m = n$, then the matrix space is an algebra, thus, is a ring.

From the point of view of Python base classes, both cases are the same:

```
sage: MS1 = MatrixSpace(QQ,2,3)
sage: isinstance(MS1, Ring)
False
sage: MS2 = MatrixSpace(QQ,2)
sage: isinstance(MS2, Ring)
False
```

Sage's category framework can differentiate the two cases:

```
sage: Rings()
Category of rings
sage: MS1 in Rings()
False
sage: MS2 in Rings()
True
```

And indeed, `MS2` has *more* methods than `MS1`:

```
sage: import inspect
sage: len([s for s in dir(MS1) if inspect.ismethod(getattr(MS1,s,None))])
57
sage: len([s for s in dir(MS2) if inspect.ismethod(getattr(MS2,s,None))])
81
```

This is because the class of `MS2` also inherits from the parent class for algebras:

```
sage: MS1.__class__.__bases__
(<class 'sage.matrix.matrix_space.MatrixSpace'>,
 <class 'sage.categories.vector_spaces.VectorSpaces.parent_class'>)
sage: MS2.__class__.__bases__
(<class 'sage.matrix.matrix_space.MatrixSpace'>,
 <class 'sage.categories.algebras.Algebras.parent_class'>)
```

Below, we will explain how this can be taken advantage of.

It is no surprise that our parent P defined above knows that it belongs to the category of fields, as it is derived from the base class of fields.

```
sage: P.category()
Category of fields
```

However, we could choose a smaller category, namely the category of quotient fields.

Why should one choose a category? One can provide **default methods** *for all objects* of a category, and *for all elements* of such objects. Hence, the category framework is a way to inherit useful stuff that is not present in the base classes. These default methods do not rely on implementation details, but on mathematical concepts.

In addition, the categories define **test suites** for their objects and elements—see the last section. Hence, one also gets basic sanity tests for free.

How does the *category framework* work? Abstract base classes for the objects ("parent_class") and the elements of objects ("element_class") are provided by attributes of the category. During initialisation of a parent, the class of the parent is *dynamically changed* into a sub–class of the category's parent class. Likewise, sub–classes of the category's element class are available for the creation of elements of the parent, as explained below.

A dynamic change of classes does not work in Cython. Nevertheless, method inheritance still works, by virtue of a `__getattr__` method.

Note: It is strongly recommended to use the category framework both in Python and in Cython.

Let us see whether there is any gain in chosing the category of quotient fields instead of the category of fields:

```
sage: QuotientFields().parent_class, QuotientFields().element_class
(<class 'sage.categories.quotient_fields.QuotientFields.parent_class'>,
 <class 'sage.categories.quotient_fields.QuotientFields.element_class'>)
sage: [p for p in dir(QuotientFields().parent_class) if p not in dir(Fields().parent_class)]
[]
sage: [p for p in dir(QuotientFields().element_class) if p not in dir(Fields().element_class)]
['_derivative', 'denominator', 'derivative', 'factor',
 'numerator', 'partial_fraction_decomposition']
```

So, there is no immediate gain for our fraction fields, but additional methods become available to our fraction field elements. Note that some of these methods are place-holders: There is no default implementation, but it is *required* (respectively is *optional*) to implement these methods:

```
sage: QuotientFields().element_class.denominator
<abstract method denominator at ...>
sage: from sage.misc.abstract_method import abstract_methods_of_class
sage: abstract_methods_of_class(QuotientFields().element_class)['optional']
['_add_', '_mul_']
sage: abstract_methods_of_class(QuotientFields().element_class)['required']
['__nonzero__', 'denominator', 'numerator']
```

Hence, when implementing elements of a quotient field, it is *required* to implement methods returning the denominator and the numerator, and a method that tells whether the element is nonzero, and in addition, it is *optional* (but certainly recommended) to provide some arithmetic methods. If one forgets to implement the required methods, the test suites of the category framework will complain—see below.

Implementing the category framework for the parent We simply need to declare the correct category by an optional argument of the field constructor, where we provide the possibility to override the default category:

```
sage: from sage.categories.quotient_fields import QuotientFields
sage: class MyFrac(MyFrac):
....:     def __init__(self, base, category=None):
....:         if base not in IntegralDomains():
....:             raise ValueError, "%s is no integral domain"%base
....:         Field.__init__(self, base, category=category or QuotientFields())
```

When constructing instances of `MyFrac`, their class is dynamically changed into a new class called `MyFrac_with_category`. It is a common sub–class of `MyFrac` and of the category's parent class:

```
sage: P = MyFrac(ZZ)
sage: type(P)
<class '__main__.MyFrac_with_category'>
sage: isinstance(P,MyFrac)
True
sage: isinstance(P,QuotientFields().parent_class)
True
```

The fraction field P inherits additional methods. For example, the base class `Field` does not have a method `sum`. But P inherits such method from the category of commutative additive monoids—see `sum()`:

```
sage: P.sum.__module__
'sage.categories.additive_monoids'
```

We have seen above that we can add elements. Nevertheless, the `sum` method does not work, yet:

```
sage: a = MyElement(P, 3, 4)
sage: b = MyElement(P, 1, 2)
sage: c = MyElement(P, -1, 2)
sage: P.sum([a, b, c])
Traceback (most recent call last):
...
NotImplementedError
```

The reason is that the `sum` method starts with the return value of `P.zero_element()`, which defaults to `P(0)` — but the conversion of integers into `P` is not implemented, yet.

Implementing the category framework for the elements Similar to what we have seen for parents, a new class is dynamically created that combines the element class of the parent's category with the class that we have implemented above. However, the category framework is implemented in a different way for elements than for parents:

- We provide the parent P (or its class) with an attribute called "`Element`", whose value is a class.
- The parent *automatically* obtains an attribute `P.element_class`, that subclasses both `P.Element` and `P.category().element_class`.

Hence, for providing our fraction fields with their own element classes, **we just need to add a single line to our class**:

```
sage: class MyFrac(MyFrac):
....:     Element = MyElement
```

This little change provides several benefits:

- We can now create elements by simply calling the parent:

  ```
  sage: P = MyFrac(ZZ)
  sage: P(1), P(2,3)
  ((1):(1), (2):(3))
  ```

- There is a method `zero_element` returning the expected result:

  ```
  sage: P.zero_element()
  (0):(1)
  ```

- The `sum` method mentioned above suddenly works:

  ```
  sage: a = MyElement(P, 9, 4)
  sage: b = MyElement(P, 1, 2)
  sage: c = MyElement(P, -1, 2)
  sage: P.sum([a,b,c])
  (36):(16)
  ```

What did happen behind the scenes to make this work? We provided `P.Element`, and thus obtain `P.element_class`, which is a *lazy attribute*. It provides a *dynamic* class, which is a sub–class of both `MyElement` defined above and of `P.category().element_class`:

```
sage: P.__class__.element_class
<sage.misc.lazy_attribute.lazy_attribute object at ...>
sage: P.element_class
```

```
<class '__main__.MyFrac_with_category.element_class'>
sage: type(P.element_class)
<class 'sage.structure.dynamic_class.DynamicMetaclass'>
sage: issubclass(P.element_class, MyElement)
True
sage: issubclass(P.element_class,P.category().element_class)
True
```

The *default* `__call__` method of P passes the given arguments to `P.element_class`, adding the argument `parent=P`. This is why we are now able to create elements by calling the parent.

In particular, these elements are instances of that new dynamic class:

```
sage: type(P(2,3))
<class '__main__.MyFrac_with_category.element_class'>
```

Note: *All* elements of P should use the element class. In order to make sure that this also holds for the result of arithmetic operations, we created them as instances of `self.__class__` in the arithmetic methods of `MyElement`.

`P.zero_element()` defaults to returning `P(0)` and thus returns an instance of `P.element_class`. Since `P.sum([...])` starts the summation with `P.zero_element()` and the class of the sum only depends on the first summand, by our implementation, we have:

```
sage: type(a)
<class '__main__.MyElement'>
sage: isinstance(a,P.element_class)
False
sage: type(P.sum([a,b,c]))
<class '__main__.MyFrac_with_category.element_class'>
```

The method `factor` provided by `P.category().element_class` (see above) simply works:

```
sage: a; a.factor(); P(6,4).factor()
(9):(4)
2^-2 * 3^2
2^-1 * 3
```

But that's surprising: The element a is just an instance of `MyElement`, but not of `P.element_class`, and its class does not know about the factor method. In fact, this is due to a `__getattr__` method defined for `sage.structure.element.Element`.

```
sage: hasattr(type(a), 'factor')
False
sage: hasattr(P.element_class, 'factor')
True
sage: hasattr(a, 'factor')
True
```

A first note on performance The category framework is sometimes blamed for speed regressions, as in trac ticket #9138 and trac ticket #11900. But if the category framework is *used properly*, then it is fast. For illustration, we determine the time needed to access an attribute inherited from the element class. First, we consider an element that uses the class that we implemented above, but does not use the category framework properly:

```
sage: type(a)
<class '__main__.MyElement'>
```

```
sage: timeit('a.factor',number=1000)      # random
1000 loops, best of 3: 2 us per loop
```

Now, we consider an element that is equal to a, but uses the category framework properly:

```
sage: a2 = P(9,4)
sage: a2 == a
True
sage: type(a2)
<class '__main__.MyFrac_with_category.element_class'>
sage: timeit('a2.factor',number=1000)     # random
1000 loops, best of 3: 365 ns per loop
```

So, *don't be afraid of using categories!*

Coercion—the basics

Theoretical background

Coercion is not just *type conversion* "Coercion" in the C programming language means "automatic type conversion". However, in Sage, coercion is involved if one wants to be able to do arithmetic, comparisons, etc. between elements of distinct parents. Hence, **coercion is not about a change of types, but about a change of parents.**

As an illustration, we show that elements of the same type may very well belong to rather different parents:

```
sage: P1 = QQ['v,w']; P2 = ZZ['w,v']
sage: type(P1.gen()) == type(P2.gen())
True
sage: P1 == P2
False
```

P_2 naturally is a sub–ring of P_1. So, it makes sense to be able to add elements of the two rings—the result should then live in P_1, and indeed it does:

```
sage: (P1.gen()+P2.gen()).parent() is P1
True
```

It would be rather inconvenient if one needed to *manually* convert an element of P_2 into P_1 before adding. The coercion system does that conversion automatically.

Not every conversion is a coercion A coercion happens implicitly, without being explicitly requested by the user. Hence, coercion must be based on mathematical rigour. In our example, any element of P_2 can be naturally interpreted as an element of P_1. We thus have:

```
sage: P1.has_coerce_map_from(P2)
True
sage: P1.coerce_map_from(P2)
Conversion map:
  From: Multivariate Polynomial Ring in w, v over Integer Ring
  To:   Multivariate Polynomial Ring in v, w over Rational Field
```

While there is a conversion from P_1 to P_2 (namely restricted to polynomials with integral coefficients), this conversion is not a coercion:

```
sage: P2.convert_map_from(P1)
Conversion map:
  From: Multivariate Polynomial Ring in v, w over Rational Field
  To:   Multivariate Polynomial Ring in w, v over Integer Ring
sage: P2.has_coerce_map_from(P1)
False
sage: P2.coerce_map_from(P1) is None
True
```

The four axioms requested for coercions

1. A coercion is a morphism in an appropriate category.

   This first axiom has two implications:

   (a) A coercion is defined on all elements of a parent.

   A polynomial of degree zero over the integers can be interpreted as an integer—but the attempt to convert a polynomial of non-zero degree would result in an error:

   ```
   sage: ZZ(P2.one())
   1
   sage: ZZ(P2.gen(1))
   Traceback (most recent call last):
   ...
   TypeError: not a constant polynomial
   ```

   Hence, we only have a *partial* map. This is fine for a *conversion*, but a partial map does not qualify as a *coercion*.

   (b) Coercions are structure preserving.

   Any real number can be converted to an integer, namely by rounding. However, such a conversion is not useful in arithmetic operations, since the underlying algebraic structure is not preserved:

   ```
   sage: int(1.6)+int(2.7) == int(1.6+2.7)
   False
   ```

   The structure that is to be preserved depends on the category of the involved parents. For example, the coercion from the integers into the rational field is a homomorphism of euclidean domains:

   ```
   sage: QQ.coerce_map_from(ZZ).category_for()
   Category of euclidean domains
   ```

2. There is at most one coercion from one parent to another

   In addition, if there is a *coercion* from P_2 to P_1, then a *conversion* from P_2 to P_1 is defined for all elements of P_2 and coincides with the coercion. Nonetheless, user-exposed maps are copies of the internally used maps whence the lack of identity between different instantiations:

   ```
   sage: P1.coerce_map_from(P2) is P1.convert_map_from(P2)
   False
   ```

   For internally used maps, the maps are identical:

   ```
   sage: P1._internal_coerce_map_from(P2) is P1._internal_convert_map_from(P2)
   True
   ```

3. Coercions can be composed

If there is a coercion $\varphi : P_1 \to P_2$ and another coercion $\psi : P_2 \to P_3$, then the composition of φ followed by ψ must yield the unique coercion from P_1 to P_3.

4. The identity is a coercion

   Together with the two preceding axioms, it follows: If there are coercions from P_1 to P_2 and from P_2 to P_1, then they are mutually inverse.

Implementing a conversion We have seen above that some conversions into our fraction fields became available after providing the attribute `Element`. However, we can not convert elements of a fraction field into elements of another fraction field, yet:

```
sage: P(2/3)
Traceback (most recent call last):
...
ValueError: Numerator and denominator must be elements of Integer Ring
```

For implementing a conversion, **the default __call__ method should (almost) never be overridden.** Instead, **we implement the method _element_constructor_**, that should return an instance of the parent's element class. Some old parent classes violate that rule—please help to refactor them!

```
sage: class MyFrac(MyFrac):
....:     def _element_constructor_(self, *args, **kwds):
....:         if len(args)!=1:
....:             return self.element_class(self, *args, **kwds)
....:         x = args[0]
....:         try:
....:             P = x.parent()
....:         except AttributeError:
....:             return self.element_class(self, x, **kwds)
....:         if P in QuotientFields() and P != self.base():
....:             return self.element_class(self, x.numerator(), x.denominator(), **kwds)
....:         return self.element_class(self, x, **kwds)
```

In addition to the conversion from the base ring and from pairs of base ring elements, we now also have a conversion from the rationals to our fraction field of $\mathbf{Z}$:

```
sage: P = MyFrac(ZZ)
sage: P(2); P(2,3); P(3/4)
(2):(1)
(2):(3)
(3):(4)
```

Recall that above, the test $1 \in P$ failed with an error. We try again and find that the error has disappeared. This is because we are now able to convert the integer 1 into P. But the containment test still yields a wrong answer:

```
sage: 1 in P
False
```

The technical reason: We have a conversion $P(1)$ of 1 into P, but this is not known as a coercion—yet!

```
sage: P.has_coerce_map_from(ZZ), P.has_coerce_map_from(QQ)
(False, False)
```

Establishing a coercion There are two main ways to make Sage use a particular conversion as a coercion:

- One can use `sage.structure.parent.Parent.register_coercion()`, normally during initialisation of the parent (see documentation of the method).
- A more flexible way is to provide a method `_coerce_map_from_` for the parent.

Let P and R be parents. If `P._coerce_map_from_(R)` returns `False` or `None`, then there is no coercion from R to P. If it returns a map with domain R and codomain P, then this map is used for coercion. If it returns `True`, then the conversion from R to P is used as coercion.

Note that in the following implementation, we need a special case for the rational field, since `QQ.base()` is not the ring of integers.

```
sage: class MyFrac(MyFrac):
....:     def _coerce_map_from_(self, S):
....:         if self.base().has_coerce_map_from(S):
....:             return True
....:         if S in QuotientFields():
....:             if self.base().has_coerce_map_from(S.base()):
....:                 return True
....:             if hasattr(S,'ring_of_integers') and self.base().has_coerce_map_from(S.ring_of_inte
....:                 return True
```

By the method above, a parent coercing into the base ring will also coerce into the fraction field, and a fraction field coerces into another fraction field if there is a coercion of the corresponding base rings. Now, we have:

```
sage: P = MyFrac(QQ['x'])
sage: P.has_coerce_map_from(ZZ['x']), P.has_coerce_map_from(Frac(ZZ['x'])), P.has_coerce_map_from(QQ)
(True, True, True)
```

We can now use coercion from $\mathbf{Z}[x]$ and from $\mathbf{Q}$ into P for arithmetic operations between the two rings:

```
sage: 3/4+P(2)+ZZ['x'].gen(), (P(2)+ZZ['x'].gen()).parent() is P
((4*x + 11):(4), True)
```

Equality and element containment Recall that above, the test $1 \in P$ gave a wrong answer. Let us repeat the test now:

```
sage: 1 in P
True
```

Why is that?

The default element containment test $x \in P$ is based on the interplay of three building blocks: conversion, coercion, and equality test.

1. Clearly, if the conversion $P(x)$ raises an error, then x can not be seen as an element of P. On the other hand, a conversion $P(x)$ can generally do very nasty things. So, the fact that $P(x)$ works without error is necessary, but not sufficient for $x \in P$.
2. If P is the parent of x, then the conversion $P(x)$ will not change x (at least, that's the default). Hence, we will have $x = P(x)$.
3. Sage uses coercion not only for arithmetic operations, but also for comparison: *If* there is a coercion from the parent of x to P, then the equality test `x==P(x)` reduces to `P(x)==P(x)`. Otherwise, `x==P(x)` will evaluate as false.

That leads to the following default implementation of element containment testing:

Note: $x \in P$ holds if and only if the test `x==P(x)` does not raise an error and evaluates as true.

If the user is not happy with that behaviour, the "magical" Python method `__contains__` can be overridden.

Coercion—the advanced parts

So far, we are able to add integers and rational numbers to elements of our new implementation of the fraction field of $\mathbf{Z}$.

```
sage: P = MyFrac(ZZ)

sage: 1/2+P(2,3)+1
(13):(6)
```

Surprisingly, we can even add a polynomial over the integers to an element of P, even though the *result lives in a new parent*, namely in a polynomial ring over P:

```
sage: P(1/2) + ZZ['x'].gen(), (P(1/2) + ZZ['x'].gen()).parent() is P['x']
((1):(1)*x + (1):(2), True)
```

In the next, seemingly more easy example, there "obviously" is a coercion from the fraction field of $\mathbf{Z}$ to the fraction field of $\mathbf{Z}[x]$. However, Sage does not know enough about our new implementation of fraction fields. Hence, it does not recognise the coercion:

```
sage: Frac(ZZ['x']).has_coerce_map_from(P)
False
```

Two obvious questions arise:

1. How / why has the new ring been constructed in the example above?
2. How can we establish a coercion from P to $\mathrm{Frac}(\mathbf{Z}[x])$?

The key to answering both question is the construction of parents from simpler pieces, that we are studying now. Note that we will answer the second question *not* by providing a coercion from P to $\mathrm{Frac}(\mathbf{Z}[x])$, but by teaching Sage to automatically construct $\mathrm{MyFrac}(\mathbf{Z}[x])$ and coerce both P and $\mathrm{Frac}(\mathbf{Z}[x])$ into it.

If we are lucky, a parent can tell how it has been constructed:

```
sage: Poly,R = QQ['x'].construction()
sage: Poly,R
(Poly[x], Rational Field)
sage: Fract,R = QQ.construction()
sage: Fract,R
(FractionField, Integer Ring)
```

In both cases, the first value returned by `construction()` is a mathematical construction, called *construction functor*—see `ConstructionFunctor`. The second return value is a simpler parent to which the construction functor is applied.

Being functors, the same construction can be applied to different objects of a category:

```
sage: Poly(QQ) is QQ['x']
True
sage: Poly(ZZ) is ZZ['x']
True
sage: Poly(P) is P['x']
True
```

```
sage: Fract(QQ['x'])
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

Let us see on which categories these construction functors are defined:

```
sage: Poly.domain()
Category of rings
sage: Poly.codomain()
Category of rings
sage: Fract.domain()
Category of integral domains
sage: Fract.codomain()
Category of fields
```

In particular, the construction functors can be composed:

```
sage: Poly*Fract
Poly[x](FractionField(...))
sage: (Poly*Fract)(ZZ) is QQ[x]
True
```

In addition, it is often assumed that we have a coercion from input to output of the construction functor:

```
sage: ((Poly*Fract)(ZZ)).coerce_map_from(ZZ)
Composite map:
  From: Integer Ring
  To:   Univariate Polynomial Ring in x over Rational Field
  Defn:   Natural morphism:
          From: Integer Ring
          To:   Rational Field
        then
          Polynomial base injection morphism:
          From: Rational Field
          To:   Univariate Polynomial Ring in x over Rational Field
```

Construction functors do not necessarily commute:

```
sage: (Fract*Poly)(ZZ)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

The pushout of construction functors We can now formulate our problem. We have parents P_1, P_2 and R, and construction functors F_1, F_2, such that $P_1 = F_1(R)$ and $P_2 = F_2(R)$. We want to find a new construction functor F_3, such that both P_1 and P_2 coerce into $P_3 = F_3(R)$.

In analogy to a notion of category theory, P_3 is called the `pushout()` of P_1 and P_2; and similarly F_3 is called the pushout of F_1 and F_2.

```
sage: from sage.categories.pushout import pushout
sage: pushout(Fract(ZZ),Poly(ZZ))
Univariate Polynomial Ring in x over Rational Field
```

$F_1 \circ F_2$ and $F_2 \circ F_1$ are natural candidates for the pushout of F_1 and F_2. However, the order of the functors must rely on a canonical choice. "Indecomposable" construction functors have a *rank*, and this allows to order them canonically:

Note: If `F1.rank` is smaller than `F2.rank`, then the pushout is $F_2 \circ F_1$ (hence, F_1 is applied first).

We have

```
sage: Fract.rank, Poly.rank
(5, 9)
```

and thus the pushout is

```
sage: Fract.pushout(Poly), Poly.pushout(Fract)
(Poly[x](FractionField(...)), Poly[x](FractionField(...)))
```

This is why the example above has worked.

However, only "elementary" construction functors have a rank:

```
sage: (Fract*Poly).rank
Traceback (most recent call last):
...
AttributeError: 'CompositeConstructionFunctor' object has no attribute 'rank'
```

Shuffling composite construction functors If composed construction fuctors ... $\circ F_2 \circ F_1$ and ... $\circ G_2 \circ G_1$ are given, then Sage determines their pushout by *shuffling* the constituents:

- If `F1.rank < G1.rank` then we apply F_1 first, and continue with ... $\circ F_3 \circ F_2$ and ... $\circ G_2 \circ G_1$.
- If `F1.rank > G1.rank` then we apply G_1 first, and continue with ... $\circ F_2 \circ F_1$ and ... $\circ G_3 \circ G_2$.

If `F1.rank == G1.rank`, then the tie needs to be broken by other techniques (see below).

As an illustration, we first get us some functors and then see how chains of functors are shuffled.

```
sage: AlgClos, R = CC.construction(); AlgClos
AlgebraicClosureFunctor
```

```
sage: Compl, R = RR.construction(); Compl
Completion[+Infinity]
```

```
sage: Matr, R = (MatrixSpace(ZZ,3)).construction(); Matr
MatrixFunctor
```

```
sage: AlgClos.rank, Compl.rank, Fract.rank, Poly.rank, Matr.rank
(3, 4, 5, 9, 10)
```

When we apply `Fract`, `AlgClos`, `Poly` and `Fract` to the ring of integers, we obtain:

```
sage: (Fract*Poly*AlgClos*Fract)(ZZ)
Fraction Field of Univariate Polynomial Ring in x over Algebraic Field
```

When we apply `Compl`, `Matr` and `Poly` to the ring of integers, we obtain:

```
sage: (Poly*Matr*Compl)(ZZ)
Univariate Polynomial Ring in x over Full MatrixSpace of 3 by 3 dense matrices over Real Field with 5
```

Applying the shuffling procedure yields

```
sage: (Poly*Matr*Fract*Poly*AlgClos*Fract*Compl)(ZZ)
Univariate Polynomial Ring in x over Full MatrixSpace of 3 by 3 dense matrices over Fraction Field o
```

and this is indeed equal to the pushout found by Sage:

```
sage: pushout((Fract*Poly*AlgClos*Fract)(ZZ), (Poly*Matr*Compl)(ZZ))
Univariate Polynomial Ring in x over Full MatrixSpace of 3 by 3 dense matrices over Fraction Field o
```

Breaking the tie If `F1.rank==G1.rank` then Sage's pushout constructions offers two ways to proceed:

1. Construction functors have a method `merge()` that either returns `None` or returns a construction functor—see below. If either `F1.merge(G1)` or `G1.merge(F1)` returns a construction functor H_1, then we apply H_1 and continue with $... \circ F_3 \circ F_2$ and $... \circ G_3 \circ G_2$.
2. Construction functors have a method `commutes()`. If either `F1.commutes(G1)` or `G1.commutes(F1)` returns `True`, then we apply both F_1 and G_1 in any order, and continue with $... \circ F_3 \circ F_2$ and $... \circ G_3 \circ G_2$.

By default, `F1.merge(G1)` returns `F1` if `F1==G1`, and returns `None` otherwise. The `commutes()` method exists, but it seems that so far nobody has implemented two functors of the same rank that commute.

Establishing a default implementation The typical application of `merge()` is to provide a coercion between *different implementations* of the *same algebraic structure*.

Note: If `F1(P)` and `F2(P)` are different implementations of the same thing, then `F1.merge(F2)(P)` should return the default implementation.

We want to boldly turn our toy implementation of fraction fields into the new default implementation. Hence:

- Next, we implement a new version of the "usual" fraction field functor, having the same rank, but returning our new implementation.
- We make our new implementation the default, by virtue of a merge method.

Warning:
- Do not override the default `__call__` method of `ConstructionFunctor`—implement `_apply_functor` instead.
- Declare domain and codomain of the functor during initialisation.

```
sage: from sage.categories.pushout import ConstructionFunctor
sage: class MyFracFunctor(ConstructionFunctor):
....:     rank = 5
....:     def __init__(self):
....:         ConstructionFunctor.__init__(self, IntegralDomains(), Fields())
....:     def _apply_functor(self, R):
....:         return MyFrac(R)
....:     def merge(self, other):
....:         if isinstance(other, (type(self), sage.categories.pushout.FractionField)):
....:             return self

sage: MyFracFunctor()
MyFracFunctor
```

We verify that our functor can really be used to construct our implementation of fraction fields, and that it can be merged with either itself or the usual fraction field constructor:

```
sage: MyFracFunctor()(ZZ)
NewFrac(Integer Ring)
```

```
sage: MyFracFunctor().merge(MyFracFunctor())
MyFracFunctor

sage: MyFracFunctor().merge(Fract)
MyFracFunctor
```

There remains to let our new fraction fields know about the new construction functor:

```
sage: class MyFrac(MyFrac):
....:     def construction(self):
....:         return MyFracFunctor(), self.base()

sage: MyFrac(ZZ['x']).construction()
(MyFracFunctor, Univariate Polynomial Ring in x over Integer Ring)
```

Due to merging, we have:

```
sage: pushout(MyFrac(ZZ['x']), Frac(QQ['x']))
NewFrac(Univariate Polynomial Ring in x over Rational Field)
```

A second note on performance Being able to do arithmetics involving elements of different parents, with the automatic creation of a pushout to contain the result, is certainly convenient—but one should not rely on it, if speed matters. Simply the conversion of elements into different parents takes time. Moreover, by trac ticket #14058, the pushout may be subject to Python's cyclic garbage collection. Hence, if one does not keep a strong reference to it, the same parent may be created repeatedly, which is a waste of time. In the following example, we illustrate the slow–down resulting from blindly relying on coercion:

```
sage: ZZxy = ZZ['x','y']
sage: a = ZZxy('x')
sage: b = 1/2
sage: timeit("c = a+b")     # random
10000 loops, best of 3: 172 us per loop
sage: QQxy = QQ['x','y']
sage: timeit("c2 = QQxy(a)+QQxy(b)") # random
10000 loops, best of 3: 168 us per loop
sage: a2 = QQxy(a)
sage: b2 = QQxy(b)
sage: timeit("c2 = a2+b2") # random
100000 loops, best of 3: 10.5 us per loop
```

Hence, if one avoids the explicit or implicit conversion into the pushout, but works in the pushout right away, one can get a more than 10–fold speed–up.

The test suites of the category framework

The category framework does not only provide functionality but also a test framework. This section logically belongs to the section on categories, but without the bits that we have implemented in the section on coercion, our implementation of fraction fields would not have passed the tests yet.

"Abstract" methods We have already seen above that a category can require/suggest certain parent or element methods, that the user must/should implement. This is in order to smoothly blend with the methods that already exist in Sage.

The methods that ought to be provided are called `abstract_method()`. Let us see what methods are needed for quotient fields and their elements:

```
sage: from sage.misc.abstract_method import abstract_methods_of_class

sage: abstract_methods_of_class(QuotientFields().parent_class)['optional']
[]
sage: abstract_methods_of_class(QuotientFields().parent_class)['required']
['__contains__']
```

Hence, the only required method (that is actually required for all parents that belong to the category of sets) is an element containment test. That's fine, because the base class `Parent` provides a default containment test.

The elements have to provide more:

```
sage: abstract_methods_of_class(QuotientFields().element_class)['optional']
['_add_', '_mul_']
sage: abstract_methods_of_class(QuotientFields().element_class)['required']
['__nonzero__', 'denominator', 'numerator']
```

Hence, the elements must provide `denominator()` and `numerator()` methods, and must be able to tell whether they are zero or not. The base class `Element` provides a default `__nonzero__()` method. In addition, the elements may provide Sage's single underscore arithmetic methods (actually any ring element *should* provide them).

The `_test_...` methods If a parent or element method's name start with "_test_", it gives rise to a test in the automatic test suite. For example, it is tested

- whether a parent P actually is an instance of the parent class of the category of P,
- whether the user has implemented the required abstract methods,
- whether some defining structural properties (e.g., commutativity) hold.

For example, if one forgets to implement required methods, one obtains the following error:

```
sage: class Foo(Parent):
....:  Element = sage.structure.element.Element
....:  def __init__(self):
....:      Parent.__init__(self, category=QuotientFields())
sage: Bar = Foo()
sage: bar = Bar.element_class(Bar)
sage: bar._test_not_implemented_methods()
Traceback (most recent call last):
...
AssertionError: Not implemented method: denominator
```

Here are the tests that form the test suite of quotient fields:

```
sage: [t for t in dir(QuotientFields().parent_class) if t.startswith('_test_')]
['_test_additive_associativity',
 '_test_an_element',
 '_test_associativity',
 '_test_characteristic',
 '_test_characteristic_fields',
 '_test_distributivity',
 '_test_elements',
 '_test_elements_eq_reflexive',
 '_test_elements_eq_symmetric',
 '_test_elements_eq_transitive',
```

```
 '_test_elements_neq',
 '_test_euclidean_degree',
 '_test_one', '_test_prod',
 '_test_quo_rem',
 '_test_some_elements',
 '_test_zero',
 '_test_zero_divisors']
```

We have implemented all abstract methods (or inherit them from base classes), we use the category framework, and we have implemented coercions. So, we are confident that the test suite runs without an error. In fact, it does!

Note: The following trick with the `__main__` module is only needed in doctests, not in an interactive session or when defining the classes externally.

```
sage: __main__.MyFrac = MyFrac
sage: __main__.MyElement = MyElement
sage: P = MyFrac(ZZ['x'])
sage: TestSuite(P).run()
```

Let us see what tests are actually performed:

```
sage: TestSuite(P).run(verbose=True)
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_category() . . . pass
running ._test_characteristic() . . . pass
running ._test_characteristic_fields() . . . pass
running ._test_distributivity() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_euclidean_degree() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_quo_rem() . . . pass
running ._test_some_elements() . . . pass
running ._test_zero() . . . pass
running ._test_zero_divisors() . . . pass
```

Implementing a new category with additional tests As one can see, tests are also performed on elements. There are methods that return one element or a list of some elements, relying on "typical" elements that can be found in most algebraic structures.

```
sage: P.an_element(); P.some_elements()
(2):(1)
[(2):(1)]
```

Unfortunately, the list of elements that is returned by the default method is of length one, and that single element could also be a bit more interesting. The method an_element relies on a method `_an_element_()`, so, we implement that. We also override the some_elements method.

```
sage: class MyFrac(MyFrac):
....:     def _an_element_(self):
....:         a = self.base().an_element()
....:         b = self.base_ring().an_element()
....:         if (a+b)!=0:
....:             return self(a)**2/(self(a+b)**3)
....:         if b != 0:
....:             return self(a)/self(b)**2
....:         return self(a)**2*self(b)**3
....:     def some_elements(self):
....:         return [self.an_element(),self(self.base().an_element()),self(self.base_ring().an_eleme
```

```
sage: P = MyFrac(ZZ['x'])
sage: P.an_element(); P.some_elements()
(x^2):(x^3 + 3*x^2 + 3*x + 1)
[(x^2):(x^3 + 3*x^2 + 3*x + 1), (x):(1), (1):(1)]
```

Now, as we have more interesting elements, we may also add a test for the "factor" method. Recall that the method was inherited from the category, but it appears that it is not tested.

Normally, a test for a method defined by a category should be provided by the same category. Hence, since `factor` is defined in the category of quotient fields, a test should be added there. But we won't change source code here and will instead create a sub–category.

Apparently, If e is an element of a quotient field, the product of the factors returned by `e.factor()` should be equal to e. For forming the product, we use the `prod` method, that, no surprise, is inherited from another category:

```
sage: P.prod.__module__
'sage.categories.monoids'
```

When we want to create a sub–category, we need to provide a method `super_categories()`, that returns a list of all immediate super categories (here: category of quotient fields).

Warning: A sub–category S of a category C is *not* implemented as a sub–class of `C.__class__`! S becomes a sub–category of C only if `S.super_categories()` returns (a sub–category of) C!

The parent and element methods of a category are provided as methods of classes that are the attributes `ParentMethods` and `Element Methods` of the category, as follows:

```
sage: from sage.categories.category import Category
sage: class QuotientFieldsWithTest(Category): # do *not* inherit from QuotientFields, but ...
....:     def super_categories(self):
....:         return [QuotientFields()]       # ... declare QuotientFields as a super category!
....:     class ParentMethods:
....:         pass
....:     class ElementMethods:
....:         def _test_factorisation(self, **options):
....:             P = self.parent()
....:             assert self == P.prod([P(b)**e for b,e in self.factor()])
```

We provide an instance of our quotient field implementation with that new category. Note that categories have a default _repr_ method, that guesses a good string representation from the name of the class: QuotientFieldsWithTest becomes "quotient fields with test".

Note: The following trick with the __main__ module is only needed in doctests, not in an interactive session or when defining the classes externally.

```
sage: __main__.MyFrac = MyFrac
sage: __main__.MyElement = MyElement
sage: __main__.QuotientFieldsWithTest = QuotientFieldsWithTest
sage: P = MyFrac(ZZ['x'], category=QuotientFieldsWithTest())
sage: P.category()
Category of quotient fields with test
```

The new test is inherited from the category. Since an_element() is returning a complicated element, _test_factorisation is a serious test:

```
sage: P.an_element()._test_factorisation
<bound method MyFrac_with_category.element_class._test_factorisation of (x^2):(x^3 + 3*x^2 + 3*x + 1)


sage: P.an_element().factor()
(x + 1)^-3 * x^2
```

Last, we observe that the new test has automatically become part of the test suite. We remark that the existing tests became more serious as well, since we made sage.structure.parent.Parent.an_element() return something more interesting.

```
sage: TestSuite(P).run(verbose=True)
running ._test_additive_associativity() . . . pass
running ._test_an_element() . . . pass
running ._test_associativity() . . . pass
running ._test_category() . . . pass
running ._test_characteristic() . . . pass
running ._test_characteristic_fields() . . . pass
running ._test_distributivity() . . . pass
running ._test_elements() . . .
  Running the test suite of self.an_element()
  running ._test_category() . . . pass
  running ._test_eq() . . . pass
  running ._test_factorisation() . . . pass
  running ._test_nonzero_equal() . . . pass
  running ._test_not_implemented_methods() . . . pass
  running ._test_pickling() . . . pass
  pass
running ._test_elements_eq_reflexive() . . . pass
running ._test_elements_eq_symmetric() . . . pass
running ._test_elements_eq_transitive() . . . pass
running ._test_elements_neq() . . . pass
running ._test_eq() . . . pass
running ._test_euclidean_degree() . . . pass
running ._test_not_implemented_methods() . . . pass
running ._test_one() . . . pass
running ._test_pickling() . . . pass
running ._test_prod() . . . pass
running ._test_quo_rem() . . . pass
running ._test_some_elements() . . . pass
```

```
running ._test_zero() . . . pass
running ._test_zero_divisors() . . . pass
```

Appendix: The complete code

```
# Importing base classes, ...
import sage
from sage.rings.ring import Field
from sage.structure.element import FieldElement
from sage.categories.category import Category
# ... the UniqueRepresentation tool,
from sage.structure.unique_representation import UniqueRepresentation
# ... some categories, and ...
from sage.categories.fields import Fields
from sage.categories.quotient_fields import QuotientFields
from sage.categories.integral_domains import IntegralDomains
# construction functors
from sage.categories.pushout import ConstructionFunctor

# Fraction field elements
class MyElement(FieldElement):
    def __init__(self, parent, n, d=None):
        if parent is None:
            raise ValueError("The parent must be provided")
        B = parent.base()
        if d is None:
            # The default denominator is one
            d = B.one_element()
        # verify that both numerator and denominator belong to the base
        if n not in B or d not in B:
            raise ValueError("Numerator and denominator must be elements of %s"%B)
        # Numerator and denominator should not just be "in" B,
        # but should be defined as elements of B
        d = B(d)
        n = B(n)
        # the denominator must not be zero
        if d==0:
            raise ZeroDivisionError("The denominator must not be zero")
        # normalize the denominator: WLOG, it shall be non-negative.
        if d<0:
            self.n = -n
            self.d = -d
        else:
            self.n = n
            self.d = d
        FieldElement.__init__(self,parent)

    # Methods required by the category of fraction fields:
    def numerator(self):
        return self.n
    def denominator(self):
        return self.d

    # String representation (single underscore!)
    def _repr_(self):
        return "(%s):(%s)"%(self.n,self.d)
```

```
52
53      # Comparison: We can assume that both arguments are coerced
54      # into the same parent, which is a fraction field. Hence, we
55      # are allowed to use the denominator() and numerator() methods
56      # on the second argument.
57      def __cmp__(self, other):
58          return cmp(self.n*other.denominator(), other.numerator()*self.d)
59      # Arithmetic methods, single underscore. We can assume that both
60      # arguments are coerced into the same parent.
61      # We return instances of self.__class__, because self.__class__ will
62      # eventually be a sub-class of MyElement.
63      def _add_(self, other):
64          C = self.__class__
65          D = self.d*other.denominator()
66          return C(self.parent(), self.n*other.denominator()+self.d*other.numerator(),D)
67      def _sub_(self, other):
68          C = self.__class__
69          D = self.d*other.denominator()
70          return C(self.parent(), self.n*other.denominator()-self.d*other.numerator(),D)
71      def _mul_(self, other):
72          C = self.__class__
73          return C(self.parent(), self.n*other.numerator(), self.d*other.denominator())
74      def _div_(self, other):
75          C = self.__class__
76          return C(self.parent(), self.n*other.denominator(), self.d*other.numerator())
77
78  # Inheritance from UniqueRepresentation implements the unique parent
79  # behaviour. Moreover, it implements pickling (provided that Python
80  # succeeds to look up the class definition).
81  class MyFrac(UniqueRepresentation, Field):
82      # Implement the category framework for elements, which also
83      # makes some basic conversions work.
84      Element = MyElement
85
86      # Allow to pass to a different category, by an optional argument
87      def __init__(self, base, category=None):
88          # Fraction fields only exist for integral domains
89          if base not in IntegralDomains():
90              raise ValueError, "%s is no integral domain"%base
91          # Implement the category framework for the parent
92          Field.__init__(self, base, category=category or QuotientFields())
93
94      # Single-underscore method for string representation
95      def _repr_(self):
96          return "NewFrac(%s)"%repr(self.base())
97
98      # Two methods that are implicitly used in some tests
99      def base_ring(self):
100         return self.base().base_ring()
101     def characteristic(self):
102         return self.base().characteristic()
103
104     # Implement conversions. Do not override __call__!
105     def _element_constructor_(self, *args, **kwds):
106         if len(args)!=1:
107            return self.element_class(self, *args, **kwds)
108         x = args[0]
109         try:
```

```
110             P = x.parent()
111         except AttributeError:
112             return self.element_class(self, x, **kwds)
113         if P in QuotientFields() and P != self.base():
114             return self.element_class(self, x.numerator(), x.denominator(), **kwds)
115         return self.element_class(self, x, **kwds)
116
117     # Implement coercion from the base and from fraction fields
118     # over a ring that coerces into the base
119     def _coerce_map_from_(self, S):
120         if self.base().has_coerce_map_from(S):
121             return True
122         if S in QuotientFields():
123             if self.base().has_coerce_map_from(S.base()):
124                 return True
125             if hasattr(S,'ring_of_integers') and self.base().has_coerce_map_from(S.ring_of_integers()
126                 return True
127     # Tell how this parent was constructed, in order to enable pushout constructions
128     def construction(self):
129         return MyFracFunctor(), self.base()
130
131     # return some elements of this parent
132     def _an_element_(self):
133         a = self.base().an_element()
134         b = self.base_ring().an_element()
135         if (a+b)!=0:
136             return self(a)**2/(self(a+b)**3)
137         if b != 0:
138             return self(a)/self(b)**2
139         return self(a)**2*self(b)**3
140     def some_elements(self):
141         return [self.an_element(),self(self.base().an_element()),self(self.base_ring().an_element())]
142
143
144 # A construction functor for our implementation of fraction fields
145 class MyFracFunctor(ConstructionFunctor):
146     # The rank is the same for Sage's original fraction field functor
147     rank = 5
148     def __init__(self):
149         # The fraction field construction is a functor
150         # from the category of integral domains into the category of
151         # fields
152         # NOTE: We could actually narrow the codomain and use the
153         # category QuotientFields()
154         ConstructionFunctor.__init__(self, IntegralDomains(), Fields())
155     # Applying the functor to an object. Do not override __call__!
156     def _apply_functor(self, R):
157         return MyFrac(R)
158     # Note: To apply the functor to morphisms, implement
159     #       _apply_functor_to_morphism
160
161     # Make sure that arithmetic involving elements of Frac(R) and
162     # MyFrac(R) works and yields elements of MyFrac(R)
163     def merge(self, other):
164         if isinstance(other, (type(self), sage.categories.pushout.FractionField)):
165             return self
166
167 # A quotient field category with additional tests.
```

```
# Notes:
# - Category inherits from UniqueRepresentation. Hence, there
#   is only one category for given arguments.
# - Since QuotientFieldsWithTest is a singleton (there is only
#   one instance of this class), we could inherit from
#   sage.categories.category_singleton.Category_singleton
#   rather than from sage.categories.category.Category
class QuotientFieldsWithTest(Category):
    # Our category is a sub-category of the category of quotient fields,
    # by means of the following method.
    def super_categories(self):
        return [QuotientFields()]

    # Here, we could implement methods that are available for
    # all objects in this category.
    class ParentMethods:
        pass

    # Here, we add a new test that is available for all elements
    # of any object in this category.
    class ElementMethods:
        def _test_factorisation(self, **options):
            P = self.parent()
            # The methods prod() and factor() are inherited from
            # some other categories.
            assert self == P.prod([P(b)**e for b,e in self.factor()])
```

BIBLIOGRAPHY

[Stanley2013] Richard Stanley. *Algebraic Combinatorics: walks, trees, tableaux and more*, Springer, first edition, 2013.

[Bidigare1997] Thomas Patrick Bidigare. *Hyperplane arrangement face algebras and their associated Markov chains*. ProQuest LLC, Ann Arbor, MI, 1997. Thesis (Ph.D.) University of Michigan.

[Brown2000] Kenneth S. Brown. *Semigroups, rings, and Markov chains*. J. Theoret. Probab., 13(3):871-938, 2000.

[AKS2013] Arvind Ayyer, Steven Klee, Anne Schilling. *Combinatorial Markov chains on linear extensions* J. Algebraic Combinatorics, doi:10.1007/s10801-013-0470-9, Arxiv 1205.7074.

[BN] Matthew Baker, Serguei Norine, Riemann-Roch and Abel-Jacobi Theory on a Finite Graph, Advances in Mathematics 215 (2007), 766–788.

[BTW] Per Bak, Chao Tang and Kurt Wiesenfeld (1987). *Self-organized criticality: an explanation of 1/f noise*, Physical Review Letters 60: 381–384 Wikipedia article.

[CRS] Robert Cori, Dominique Rossin, and Bruno Salvy, *Polynomial ideals for sandpiles and their Gröbner bases*, Theoretical Computer Science, 276 (2002) no. 1–2, 1–15.

[H] Holroyd, Levine, Meszaros, Peres, Propp, Wilson, Chip-Firing and Rotor-Routing on Directed Graphs. The final version of this paper appears in *In and out of Equilibrium II*, Eds. V. Sidoravicius, M. E. Vares, in the Series Progress in Probability, Birkhauser (2008).

[Bourbaki46] Nicolas Bourbaki. *Lie Groups and Lie Algebras: Chapters 4-6*. Springer, reprint edition, 1998.

[BumpNakasuji2010] D. Bump and M. Nakasuji. Casselman's basis of Iwahori vectors and the Bruhat order. arXiv:1002.2996, http://arxiv.org/abs/1002.2996.

[Carrell1994] J. B. Carrell. The Bruhat graph of a Coxeter group, a conjecture of Deodhar, and rational smoothness of Schubert varieties. In *Algebraic Groups and Their Generalizations: Classical Methods*, AMS Proceedings of Symposia in Pure Mathematics, 56, 53–61, 1994.

[Deodhar1977] V. V. Deodhar. Some characterizations of Bruhat ordering on a Coxeter group and determination of the relative Moebius function. Inventiones Mathematicae, 39(2):187–198, 1977.

[Dyer1993] M. J. Dyer. The nil Hecke ring and Deodhar's conjecture on Bruhat intervals. Inventiones Mathematicae, 111(1):571–574, 1993.

[Dynkin1952] E. B. Dynkin, Semisimple subalgebras of semisimple Lie algebras. (Russian) Mat. Sbornik N.S. 30(72):349–462, 1952.

[FauserEtAl2006] B. Fauser, P. D. Jarvis, R. C. King, and B. G. Wybourne. New branching rules induced by plethysm. *Journal of Physics A*. 39(11):2611–2655, 2006.

[Fulton1997] W. Fulton. *Young Tableaux*. Cambridge University Press, 1997.

[FourierEtAl2009] G. Fourier, M. Okado, A. Schilling. Kirillov–Reshetikhin crystal for nonexceptional types. *Advances in Mathematics*, 222:1080–1116, 2009.

[FourierEtAl2010] G. Fourier, M. Okado, A. Schilling. Perfectness of Kirillov-Reshetikhin crystals for nonexceptional types. *Contemp. Math.*, 506:127–143, 2010.

[HatayamaEtAl2001] G. Hatayama, A. Kuniba, M. Okado, T. Takagi, Z. Tsuboi. Paths, crystals and fermionic formulae. in MathPhys Odyssey 2001, in : Prog. Math. Phys., vol 23, Birkhauser Boston, Boston, MA 2002, pp. 205–272.

[HainesEtAl2009] T. J. Haines, R. E. Kottwitz, and A. Prasad. Iwahori-Hecke Algebras. arXiv:math/0309168, http://arxiv.org/abs/math/0309168.

[HongKang2002] J. Hong and S.-J. Kang. *Introduction to Quantum Groups and Crystal Bases*. AMS Graduate Studies in Mathematics, American Mathematical Society, 2002.

[HoweEtAl2005] R. Howe, E.-C.Tan, and J. F. Willenbring. Stable branching rules for classical symmetric pairs. *Transactions of the American Mathematical Society*, 357(4):1601–1626, 2005.

[Iwahori1964] N. Iwahori. On the structure of a Hecke ring of a Chevalley group over a finite field. *J. Fac. Sci. Univ. Tokyo Sect. I*, 10:215–236, 1964.

[Jimbo1986] M. A. Jimbo. q-analogue of $U(\mathfrak{gl}(N+1))$, Hecke algebra, and the Yang-Baxter equation. *Lett. Math. Phys*, 11(3):247–252, 1986.

[JonesEtAl2010] B. Jones, A. Schilling. Affine structures and a tableau model for E_6 crystals *J. Algebra*, 324:2512-2542, 2010

[Kac] Victor G. Kac. *Infinite Dimensional Lie algebras* Cambridge University Press, third edition, 1994.

[Kashiwara1995] M. Kashiwara. On crystal bases. Representations of groups (Banff, AB, 1994), 155–197, CMS Conference Proceedings, 16, American Mathematical Society, Providence, RI, 1995.

[KashiwaraNakashima1994] M. Kashiwara and T. Nakashima. Crystal graphs for representations of the q-analogue of classical Lie algebras. *Journal Algebra*, 165(2):295–345, 1994.

[King1975] R. C. King. Branching rules for classical Lie groups using tensor and spinor methods. *Journal of Physics A*, 8:429–449, 1975.

[Knuth1970] D. Knuth. Permutations, matrices, and generalized Young tableaux. *Pacific Journal of Mathematics*, 34(3):709–727, 1970.

[Knuth1998] D. Knuth. *The Art of Computer Programming. Volume 3. Sorting and Searching*. Addison Wesley Longman, 1998.

[KKMMNN1992] S-J. Kang, M. Kashiwara, K. C. Misra, T. Miwa, T. Nakashima, A. Nakayashiki. Affine crystals and vertex models. *Int. J. Mod. Phys.* A 7 (suppl. 1A): 449–484, 1992.

[LNSSS14I] C. Lenart, S. Naito, D. Sagaki, A. Schilling, and M. Shimozono. A uniform model for for Kirillov-Reshetikhin crystals I: Lifting the parabolic quantum Bruhat graph. (2014) Arxiv 1211.2042

[LNSSS14II] C. Lenart, S. Naito, D. Sagaki, A. Schilling, and M. Shimozono. A uniform model for for Kirillov-Reshetikhin crystals II: Alcove model, path model, and $P = X$. (2014) Arxiv 1402.2203

[L1995] P. Littelmann. *Paths and root operators in representation theory*. Ann. of Math. (2) 142 (1995), no. 3, 499-525.

[McKayPatera1981] W. G. McKay and J. Patera. *Tables of Dimensions, Indices and Branching Rules for Representations of Simple Lie Algebras*. Marcel Dekker, 1981.

[KKS2007] S.-J. Kang, J.-A. Kim, and D.-U. Shin. *Modified Nakajima monomials and the crystal $B(\infty)$*. J. Algebra, **308** (2007), 524-535.

[OkadoSchilling2008] M. Okado, A.Schilling. Existence of crystal bases for Kirillov–Reshetikhin crystals for nonexceptional types. *Representation Theory* 12:186–207, 2008.

[Seitz1991] G. Seitz, Maximal subgroups of exceptional algebraic groups. Mem. Amer. Math. Soc. 90 (1991), no. 441.

[Rubenthaler2008] H. Rubenthaler, The (A2,G2) duality in E6, octonions and the triality principle. Trans. Amer. Math. Soc. 360 (2008), no. 1, 347–367

[SchillingTingley2011] A. Schilling, P. Tingley. Demazure crystals, Kirillov-Reshetikhin crystals, and the energy function. preprint arXiv:1104.2359

[Stanley1999] R. P. Stanley. *Enumerative Combinatorics, Volume 2*. Cambridge University Press, 1999.

[Testerman1989] Testerman, Donna M. A construction of certain maximal subgroups of the algebraic groups E6 and F4. J. Algebra 122 (1989), no. 2, 299–322.

[Testerman1992] Testerman, Donna M. The construction of the maximal A1's in the exceptional algebraic groups. Proc. Amer. Math. Soc. 116 (1992), no. 3, 635–644.

[CormenEtAl2001] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, USA, 2nd edition, 2001.

[MenezesEtAl1996] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, USA, 1996.

[Stinson2006] D. R. Stinson. *Cryptography: Theory and Practice*. Chapman & Hall/CRC, Boca Raton, USA, 3rd edition, 2006.

[TrappeWashington2006] W. Trappe and L. C. Washington. *Introduction to Cryptography with Coding Theory*. Pearson Prentice Hall, Upper Saddle River, New Jersey, USA, 2nd edition, 2006.