Algebraic Combinatorics Tutorial

last edited Jun 5, 2014 2:52:29 PM by admin

Combinatorics and Sage Tutorial

by Andrew Johnson

edited by Sara Billey

This worksheet is an introduction to the mathematical field of Combinatorics and an exploration of the combinat library in Sage. Combinatorics has many areas of study and so this tutorial will try to simply get a good foundation. To see the Combinatorics library for Sage go to http://www.sagemath.org/doc/reference/combinat/index.html .

For more general background on Sage see http://wstein.org/books/sagebook/sagebook.pdf and http://docs.python.org/tutorial/ and http://www.diveintopython.net/ and http://interact.sagemath.org/

Sets

We will start with the Sets data structure and its various functions in Sage even though it is not part of the combinat library. In many combinatorial proofs we are often intersted in constructing various sets, counting their elements, and/or creating bijections between 2 or more sets. For this a good understanding of the set implementation in sage will be useful. To create a set one just simply needs to pass a list/tuple or even other sets to the set() command. Reference page for Sage sets http://www.sagemath.org/doc/reference/sage/sets/set.html .

```
List1 = [1,2,3,10,'a','b','c',pi,3,2,1]
```

```
List1
```

```
[1, 2, 3, 10, 'a', 'b', 'c', pi, 3, 2, 1]
```

```
Set1 = Set(List1)
Set1
```

```
{'a', 1, 2, 3, 10, 'c', pi, 'b'}
```

Note that the sets can contain elements that differ in data types. Sage has implementation for all the basic set operations.

```
A = Set(range(10)); print A
B = Set(range(5,15)); print B

#Intersection
print "Intersection:", A & B          #or A.intersection(B)

#Union
print "Union:", A | B                 #or A.union(B)

#Difference
print "Difference A \ B:", A.difference(B)
print "Difference B \ A:", B.difference(A)
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
{5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
Intersection: {8, 9, 5, 6, 7}
Union: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
Difference A \ B: {0, 1, 2, 3, 4}
Difference B \ A: {10, 11, 12, 13, 14}
```

You can get a generator for all the subsets for any set as well as the cartesian product for two given sets.

```
C = Set([1,2,3])
D = Set([4,5,6])
sub = subsets(C)
print "subsets C:", list(sub)

Prod = CartesianProduct(C,D)
print "Cartesian Product C and D:", list(Prod)
```

```
    subsets C: [[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
    Cartesian Product C and D: [[1, 4], [1, 5], [1, 6], [2, 4], [2, 5], [2,
    6], [3, 4], [3, 5], [3, 6]]
```

To get help on a command, type its name and then "?".   To see the internal code for a command type the name of the command and "??".

```
subset?
```

```
    No object 'subset' currently defined.
```

```
subset??
```

```
No object subset
```

```
#help for commands on defined objects works the same way
C?
```

File: /Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-packages/sage/sets/set.py

Type: <class 'sage.sets.set.Set_object_enumerated_with_category'>

Definition: C(x=0, *args, **kwds)

Docstring:

```
    A finite enumerated set.
```

```
print A.cardinality()
print Prod.cardinality()
print Set(ZZ).cardinality() ## ZZ is the Integer ring (or simply the set of all integers)
```

```
    10
    9
    +Infinity
```

Counting Functions

Put simply, Combinatorics is the study of counting objects, as such there are many tools that we can use to count them. It is sometimes the case that certain number patterns appear frequently for different situations or they are used often enough that they become major fields of study. Luckily Sage has implementation for many of these Combinatorial functions, but lets first start with some basic counting principles.

The factorial of a non-negative integer n, denoted by n!, is the product of all the positive integers less than or equal to n. For example 5!=5*4*3*2*1**=**120. To calculate the factorial of n in Sage just use the factorial(n) command. By definition we say 0! = 1. An instance of this is given the set of letters {a,b,c,d,e} how many different "words" (for example adbce) can I create? The first letter has 5 choices, the second 4, the third 3, and so on and we see that it is 5!. This is an example of permutations which will be discussed later.

```
print factorial(5)
print factorial(14)
print factorial(52)
print factorial(0)
print factorial(-1)              #Gives an error
```

```
    120
    87178291200
```

```
80658175170943878571660636856403766975289505440883277824000000000000
1
Traceback (click to the left of this block for traceback)
...
ValueError: factorial -- self = (-1) must be nonnegative
```

It is often that case that you do not want to multiple all of the integers less than or equal to n, but only the last k integers (n,n-1,n-2,...,n-k+1). You may have noticed that this can be calculated by n!/(n-k)! which is simple enough but sage has no direct way to compute this like factorial(). The simple approach would be to calculate factorial(n) and factorial(n-k) then perform division, but if our k was small such as 2 this can be inefficient. So we will need to use a different function, name the falling_factorial(n,k). To continue with our word example from above for k=3 this is the same as asking how many words can we get from {a,b,c,d,e} that have length 3 to which the answer is 5*4*3 = 60 ways.

```
print falling_factorial(10,3)
print factorial(10)/factorial(10-3)
print falling_factorial(10,0)
print falling_factorial(10,10)
```

```
720
720
1
3628800
```

Binomial coefficients are another set of commonly used numbers and are very similar to our last function. For positive inters n,k where k is less than or equal to n the binomial coefficient of n and k is n!/[k!(n-k)!], this looks very similar to our previous function but with an extra k! in the divisor. An easy way to see why the k! is needed is to again look at our word problem, note that 'abd' and 'bad' have the same letters but in different order. But if we were more interested in what elements were *chosen* and ignoring the order in which they were chosen then 'abd' and 'bad are the same, but our falling_factorial() function will count both of these words which is too many and so we must divide by the number of ways we can order a word of size k, which we already found to be k!

There are two functions in Sage that handle binomial coffecients, binomial(n,k) and binomial_coefficients(n). The first function returns a single solution for specific values of n and k whereas the second function returns a dictionary of all the binimial coefficients j and k such that j+k = n.

```
print binomial(8,2)
print binomial_coefficients(10)
```

```
28
{(7, 3): 120, (4, 6): 210, (9, 1): 10, (6, 4): 210, (2, 8): 45, (5, 5):
252, (0, 10): 1, (1, 9): 10, (3, 7): 120, (10, 0): 1, (8, 2): 45}
```

Next we have the famous Fibonacci numbers which are defined as the following recurrence relation: For a function F let F(0)=0, F(1)=1 and F(n)=F(n-1)+F(n-2). In Sage this can be found using fibonacci(n) or if you want to go through the first n fibonacci numbers use fibonacci_sequence(n) which returns a generator and you can use the .next() method to get each element. The Fibonacci numbers and binomial coefficients have many interesting relationships with each other which can be found here for those interested.

```
print fibonacci(5)
print fibonacci(50)
G = fibonacci_sequence(10)
for i in range(10):
    print G.next()
```

```
5
12586269025
0
1
1
2
3
5
8
13
21
34
```

Other combinatorial functions of interest are the Catalan numbers, Stirling (first/second kind) numbers, Euler numbers, and Bell numbers. We will see later that the Stirling numbers and Bell numbers have their roots in set partitions.

Permutations

A permutation on a set of objects is an arrangement of those objects in some order. In the combinat library for Sage there is conviently a Permutation class that can perform many operations as needed, see it here. An example of a permutation is 4123 from the set {1,2,3,4}. To get a list of all the permutations of a set you can call Permutations(n) where n is an integer, list, set, or string. The elements of the list are of type Permutation_Class.

```
print Permutations([1,2,3]).list(), "\n"
print Permutations(4).list(), "\n"            #all the Permutations of the set {1,2,3,4}
print Permutations(['a',pi, 1]).list(), "\n"
print Permutations("Hey").list()
```

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

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

    [['a', pi, 1], ['a', 1, pi], [pi, 'a', 1], [pi, 1, 'a'], [1, 'a', pi],
    [1, pi, 'a']]

    [['H', 'e', 'y'], ['H', 'y', 'e'], ['e', 'H', 'y'], ['e', 'y', 'H'],
    ['y', 'H', 'e'], ['y', 'e', 'H']]
```

The individual elements of these lists have data type Permutation_Class that also have many useful functions we will explore later.

What the function actually returns is a generator G which will produce the desired permutations as necessary. Many combinatorial commands in Sage return generators and knowing how to iterate through these lists is important. One way to get these elements is with the G.list() command which returns a list of all the elements in the generator but this can be terribly inefficient for large lists. Another way to retrieve elements is to use list syntax g[i] (zero-based indexing) and in combination with G.cardinality() it is a good way to iterate over the elements without converting G to a list.

```
G = Permutations(3)
print type(G[0])
print G.list()
for i in range(G.cardinality()):
    print G[i]
```

```
    <class 'sage.combinat.permutation.Permutation_class'>
    [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    [1, 2, 3]
    [1, 3, 2]
    [2, 1, 3]
    [2, 3, 1]
    [3, 1, 2]
    [3, 2, 1]
```

While the above method is valid it can be inefficient if the cardinality method is hard to calculate. Generators have a command G.next(obj) that takes the input object and outputs the next object that would occur in the list. If the object is not part of the generator or it is at the end of the list it will return either None or False. Similarily there are G.first() and G.last() commands that get the first and last elements. We will perform the above code again but using these methods.

```
p = G.first()
while(p):
    print p
    p = G.next(p)
```

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

For a set with n elements, the total number of permutations can be found by first choosing an element to take the first spot which can be done in n ways, for the second element n-1 ways, and so on and therefore the total number of permutations is n*(n-1)*(n-2)...*1 = factorial(n). Any set that is an ordering on n elements is called the symmetric group, S_n, of that set. The cardinality of any S_n is factorial(n) for any set of n elements. For this tutorial define [n] to be the set of the first n integers {1,2,...,n}.

```
L = Permutations(8)                #Permutations of [8]
L.cardinality() == factorial(8)
```

```
    True
```

Sometimes we are only interested in permutations of the n elements that have a certain length k. To do so just pass you desired length k as a second argument: Permutations(n,k). If you recall from the Counting Functions section of this tutorial we used a falling_factorial function, it should be readily seen that the number of k-permutations on n elements is precisely the same as the value we get from our function for the same values of n and k.

```
L = Permutations(5,3)
print L.list()                                         #Permutations of [5] with length of 3
print falling_factorial(5,3) == L.cardinality()
```

```
    [[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 2], [1, 3, 4], [1, 3, 5], [1,
    4, 2], [1, 4, 3], [1, 4, 5], [1, 5, 2], [1, 5, 3], [1, 5, 4], [2, 1, 3],
    [2, 1, 4], [2, 1, 5], [2, 3, 1], [2, 3, 4], [2, 3, 5], [2, 4, 1], [2, 4,
    3], [2, 4, 5], [2, 5, 1], [2, 5, 3], [2, 5, 4], [3, 1, 2], [3, 1, 4],
    [3, 1, 5], [3, 2, 1], [3, 2, 4], [3, 2, 5], [3, 4, 1], [3, 4, 2], [3, 4,
    5], [3, 5, 1], [3, 5, 2], [3, 5, 4], [4, 1, 2], [4, 1, 3], [4, 1, 5],
    [4, 2, 1], [4, 2, 3], [4, 2, 5], [4, 3, 1], [4, 3, 2], [4, 3, 5], [4, 5,
    1], [4, 5, 2], [4, 5, 3], [5, 1, 2], [5, 1, 3], [5, 1, 4], [5, 2, 1],
    [5, 2, 3], [5, 2, 4], [5, 3, 1], [5, 3, 2], [5, 3, 4], [5, 4, 1], [5, 4,
    2], [5, 4, 3]]
    True
```

To create an individual element of a Permutation_Class object you can simply do the following.

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

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

Permutation_Class objects have many useful methods to learn details about the permutation you created. For example when examing permutations of [n] it is possible to get the number of fixed points or the number of peaks in a permutation. A fixed point in combinatorics is that for some integer $0<i<n+1$, the i$_{th}$ spot in the permutation is i, in 1342 the number 1 is in the first spot and so it is a fixed point. A peak in a permutation is when three consecutive numbers ijk have the property that $i<j>k$.

```
print p.number_of_fixed_points()
print p.number_of_peaks()
```

```
2
1
```

For those more knowledgeble in the field you can even get a list of permutations that avoid certain patterns or check if a permutation does avoid a pattern. It is obvious that if we remove all the permutations that have a pattern then we will have fewer elements then that of the symmetric group of n.

```
L = Permutations(5, avoiding = [3,4,1,2])
print L
print L.cardinality()
L.cardinality() < factorial(5)
```

```
Standard permutations of 5 avoiding [[3, 4, 1, 2]]
103
True
```

```
p = Permutation([1,2,3,5,4])
print p.avoids([3,1,2,4])
print p.avoids([1,2,3,4])
```

```
True
False
```

Set Partitions

A partition of a set X divides or breaks up X into smaller sets in such a way that no two sets have a common element and the union of all of these sets gives us back the original X. For example consider the set {1,2,3}, this set can be broken into the following two parts {1,3} and {2}. This of course is not the only way to break up this set into two parts. To get all the partitions of a set in Sage use the SetPartitions(n) where n is an integer, string, list, or a set.

```
A = SetPartitions(3)            #Partitions of [3]
print A.list()
print A.cardinality(), "\n"

B = SetPartitions([2,3,4,5])
print B.list()
print B.cardinality()
```

```
[{{1, 2, 3}}, {{2, 3}, {1}}, {{1, 3}, {2}}, {{1, 2}, {3}}, {{2}, {3},
{1}}]
5

[{{2, 3, 4, 5}}, {{2}, {3, 4, 5}}, {{2, 4, 5}, {3}}, {{2, 3, 5}, {4}},
{{5}, {2, 3, 4}}, {{2, 3}, {4, 5}}, {{2, 4}, {3, 5}}, {{3, 4}, {2, 5}},
{{2}, {3}, {4, 5}}, {{2}, {4}, {3, 5}}, {{3, 4}, {5}, {2}}, {{4}, {3},
{2, 5}}, {{2, 4}, {5}, {3}}, {{5}, {4}, {2, 3}}, {{4}, {5}, {2}, {3}}]
15
```

If you are interested only in partitions of k many parts then you can simply do the following.

```
P = SetPartitions(4,3)        #Partitions of [4] into 3 parts
print type(P[0])
print P[0], "\n"
print P.list()
print P.cardinality()
```

```
<class 'sage.sets.set.Set_object_enumerated_with_category'>
{{3, 4}, {2}, {1}}

[{{3, 4}, {2}, {1}}, {{2, 4}, {3}, {1}}, {{4}, {2, 3}, {1}}, {{1, 4},
{2}, {3}}, {{1, 3}, {4}, {2}}, {{1, 2}, {4}, {3}}]
6
```

If you wish to be really specific on how you partition your set you can pass as a second arguement a list instead of an integer. For example if you want to partition [5] into 3 sets such that one set has 3 elements and the other two have 1 element pass the list [3,1,1]. It is important that the list be in descending order otherwise you will get an error.

```
print SetPartitions(5,[3,1,1]).list()
print SetPartitions(5,[3,1,1]).cardinality()
```

```
[{{2}, {3, 4, 5}, {1}}, {{2, 4, 5}, {3}, {1}}, {{2, 3, 5}, {4}, {1}},
{{5}, {2, 3, 4}, {1}}, {{1, 4, 5}, {2}, {3}}, {{4}, {1, 3, 5}, {2}},
{{1, 3, 4}, {5}, {2}}, {{4}, {3}, {1, 2, 5}}, {{5}, {3}, {1, 2, 4}},
{{4}, {5}, {1, 2, 3}}]
10
```

There are two combinatorial functions that are concerned with set partitions, Stirling numbers of the second kind and Bell numbers. The Stirling numbers count the number of partitions of [n] into k parts while Bell numbers counts all the possible partitions of [n]. In Sage these numbers can be retrieved with stirling_number2 and bell_number. To show these equalities we will count the elements ourselves.

```
count = 0
A = SetPartitions(6,3)                                      #Partitions of [10] into 3 parts
g = A.first()
while(g):
    count+=1
    g = A.next(g)
print stirling_number2(6,3) == count

count = 0
B = SetPartitions(6)                                        #Partitions of [10]
g = B.first()
while(g):
    count+=1
    g = B.next(g)
print bell_number(6) == count
```

```
True
True
```

```
print stirling_number2(6,3)
print bell_number(6)
```

```
90
203
```

Integer Partitions

Partitioning an integer n is to find a list of positive integers so that their sum is n. For example [3,1] is a partition of 4 since 3+1=4 and so is [2,2]. The lists [3,1] and [1,3] are the same partitions and so the lists are are required to be in decreasing order so as to prevent multiples of the same partition. The Partitions class in Sage can easily handle our partitioning needs and it is more extensive then the SetPartitions class.

```
number_of_partitions(100)  # this is pretty fast!
```

```
190569292
```

```
P = Partitions(4)
print P.list()
print P.cardinality()
print P[0]
print type(P)
print type(P[0])
```

```
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
5
[4]
```

```
<class 'sage.combinat.partition.Partitions_n_with_category'>
<class
'sage.combinat.partition.Partitions_n_with_category.element_class'>
```

If you want partitions of a specific length (the number of parts) you must do the following

```
print Partitions(7,length = 3).list()
print Partitions(7,length = 3).cardinality()
print type(Partitions(7,length = 3))
```

```
[[5, 1, 1], [4, 2, 1], [3, 3, 1], [3, 2, 2]]
4
<class 'sage.combinat.integer_list.IntegerListsLex_with_category'>
```

```
print Partitions(7, min_length = 2).list(), "\n"              #Partitions of at least length 2
print Partitions(7, max_length = 5).list(), "\n"              #Partitions of at most length 5
print Partitions(7, min_length = 2, max_length = 5).list()    #Partitions with length between 2 and 5
```

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

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

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

```
print Partitions(7, min_part= 2).list(), "\n"              #Partitions with parts of size at least 2
print Partitions(7, max_part = 5).list(), "\n"             #Partitions with parts of size at most 5
print Partitions(7, min_part = 2, max_part = 5).list()     #Partitions with parts of size between 2 and 5
```

```
[[7], [5, 2], [4, 3], [3, 2, 2]]

[[5, 2], [5, 1, 1], [4, 3], [4, 2, 1], [4, 1, 1, 1], [3, 3, 1], [3, 2,
2], [3, 2, 1, 1], [3, 1, 1, 1, 1], [2, 2, 2, 1], [2, 2, 1, 1, 1], [2, 1,
1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1]]

[[5, 2], [4, 3], [3, 2, 2]]
```

If the ordering of the parts is important you can instead use OrderedPartitions(). The arguements however are different and you can only specify how many parts are in your partition, default being None.

```
print OrderedPartitions(5).list(), "\n"
print OrderedPartitions(5,2).list()
```

```
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [2, 1, 1,
1], [1, 4], [1, 3, 1], [1, 2, 2], [1, 2, 1, 1], [1, 1, 3], [1, 1, 2, 1],
[1, 1, 1, 2], [1, 1, 1, 1, 1]]

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

There are otherways to specify how you wish to construct your partitions but I will not go over them. If you wish to work with a specific partition you can use the Partition(n) method where n is a list in decreasing order to create a Partition_Class object which also has many useful functions. A couple of them are the ferrers_diagram() and conjugate() methods. The Ferrers diagram for a partition is a diagram so that each part a_i has a_i boxes (or in Sage *'s) in the ith row. The conjugate of a partition p is the partition whose Ferrer's diagram is the reflection across the diagnol of p's Ferrer's diagram.

```
p = Partition([4,3,2])
print p
print p.ferrers_diagram(), "\n"
print p.conjugate()
print p.conjugate().ferrers_diagram()
```

```
[4, 3, 2]
****
***
**

[3, 3, 2, 1]
***
***
**
*
```

It is important to note that ferrers_diagram.() returns a string containing new line chars.

```
Partition([4,3,2]).ferrers_diagram()
```

```
'****\n***\n**'
```

Integer Compositions

Integer compositions are very similar to integer partitions. A composition of n is a sequence of positive integers such that their sum is n and the ordering of the integers matter, so it is very much like the OrderedPartitions mentioned preiously. If there is one or more zero elements in the sequence then it is called a weak composition of n. Similarily as for partitions you can designate any constraints you wish in your compositions.

```
c = Composition([1,3,2,1,5])
print c
print type(c), "\n"
L = Compositions(5)
print L.list()
print L.cardinality()
print type(L), "\n"
M = Compositions(5,length=2)
print M.list()
print M.cardinality()
print type(M)
```

```
[1, 3, 2, 1, 5]
<class
'sage.combinat.composition.Compositions_all_with_category.element_class'\
>

[[1, 1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 3], [1, 2, 1, 1],
[1, 2, 2], [1, 3, 1], [1, 4], [2, 1, 1, 1], [2, 1, 2], [2, 2, 1], [2,
3], [3, 1, 1], [3, 2], [4, 1], [5]]
16
<class 'sage.combinat.composition.Compositions_n_with_category'>

[[4, 1], [3, 2], [2, 3], [1, 4]]
4
<class 'sage.combinat.integer_list.IntegerListsLex_with_category'>
```

```
print Compositions(5, min_length=2).list(), "\n"
print Compositions(5, max_length=2).list()
```

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

[[5], [4, 1], [3, 2], [2, 3], [1, 4]]
```

The number of compositions of n into k non-empty parts is binomial(n-1,k-1) and the total number of compositions is 2^{n-1} , which can be proven easily by taking the summation of binomial(n-1,k-1) from k=1 to k=n and using binomial coefficient identities.

```
print Compositions(10, length = 5).cardinality() == binomial(10-1,5-1)
print Compositions(10).cardinality() == 2^(10-1)
```

```
True
True
```

If you wish to be really specific on how you partition your set you can pass as a second arguement a list instead of an integer. For example if you want to partition [5] into 3 sets such that one set has 3 elements and the other two have 1 element pass the list [3,1,1]. It is important that the list be in descending order otherwise you will get an error.

```
print SetPartitions(5,[3,1,1]).list()
print SetPartitions(5,[3,1,1]).cardinality()
```

```
[{{2}, {3, 4, 5}, {1}}, {{2, 4, 5}, {3}, {1}}, {{2, 3, 5}, {4}, {1}},
{{5}, {2, 3, 4}, {1}}, {{1, 4, 5}, {2}, {3}}, {{4}, {1, 3, 5}, {2}},
{{1, 3, 4}, {5}, {2}}, {{4}, {3}, {1, 2, 5}}, {{5}, {3}, {1, 2, 4}},
{{4}, {5}, {1, 2, 3}}]
10
```

Symmetric Functions

A symmetric function is a power series in countably many variables which is invariant under transposition of any two variables. The symmetric functions form a ring under usual addition and multiplication of power series.   There are several important bases for this ring.   Here is a brief start computing symmetric function expansions.

```
Sym=SymmetricFunctions(QQ)
```

```
Sym.inject_shorthands()
```

```
/Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-\
packages/sage/combinat/sf/sf.py:1075: RuntimeWarning: redefining global
value `h`
  inject_variable(shorthand, getattr(self, shorthand)())
/Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-\
packages/sage/combinat/sf/sf.py:1075: RuntimeWarning: redefining global
value `s`
  inject_variable(shorthand, getattr(self, shorthand)())
/Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-\
packages/sage/combinat/sf/sf.py:1075: RuntimeWarning: redefining global
value `m`
  inject_variable(shorthand, getattr(self, shorthand)())
/Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-\
packages/sage/combinat/sf/sf.py:1075: RuntimeWarning: redefining global
value `p`
  inject_variable(shorthand, getattr(self, shorthand)())
```

```
SymmetricFunctions?
```

File: /sagenb/sage_install/sage-5.11-boxen-x86_64-Linux/local/lib/python2.7/site-packages/sage/combinat/sf/sf.py

Type: <type ‘sage.misc.classcall_metaclass.ClasscallMetaclass’>

Definition: SymmetricFunctions(x=0, *args, **kwds)

Docstring:

The abstract algebra of commutative symmetric functions

Symmetric Functions in Sage

This document is an introduction to working with symmetric function theory in Sage. It is not intended to be an introduction to the theory of symm
reader is also expected to be familiar with Sage.

The algebra of symmetric functions

The algebra of symmetric functions is the unique free commutative graded connected algebra over the given ring, with one generator in each deg
algebras) of the algebra of symmetric polynomials in n variables as $n \to \infty$. Sage allows us to construct the algebra of symmetric functions ove

```
sage: Sym = SymmetricFunctions(QQ)
sage: Sym
Symmetric Functions over Rational Field
```

Sage knows certain categorical information about this algebra:

```
sage: Sym.category()
Join of Category of graded hopf algebras over Rational Field and Category of monoids with realizations and
```

Notice that `Sym` is an *abstract* algebra. This reflects the fact that there is no canonical choice for a basis. To work with specific elements, we neec

An example basis - power sums

Here is an example of how one might use the power sum realization:

```
sage: p = Sym.powersum()
sage: p
Symmetric Functions over Rational Field in the powersum basis
```

`p` now represents the realization of the symmetric function algebra on the power sum basis. The basis itself is accessible through:

```
sage: p.basis()
Lazy family (Term map from Partitions to Symmetric Functions over Rational Field in the powersum basis(i))
sage: p.basis().keys()
Partitions
```

This last line means that `p.basis()` is an association between the set of Partitions and the basis elements of the algebra `p`. To construct a spec

```
sage: p.basis()[Partition([2,1,1])]
p[2, 1, 1]
```

As this is rather cumbersome, realizations of the symmetric function algebra allow for the following abuses of notation:

```
sage: p[Partition([2, 1, 1])]
p[2, 1, 1]
sage: p[[2, 1, 1]]
p[2, 1, 1]
sage: p[2, 1, 1]
p[2, 1, 1]
```

or even:

```
sage: p[(i for i in [2, 1, 1])]
p[2, 1, 1]
```

In the special case of the empty partition, due to a limitation in Python syntax, one cannot use:

```
sage: p[]       # todo: not implemented
```

Please use instead:

```
sage: p[[]]
p[]
```

Note

When elements are constructed using the `p[something ]` syntax , an error will be raised if the input cannot be interpreted as a partition. This is

```
sage: p['something']
Traceback (click to the left of this block for traceback)
...
```

```
e(s[2,2])
```

```
    e[2, 2] - e[3, 1]
```

```
s(f)   #  f was defined under Stanley symmetric functions
```

```
    s[3, 1]
```

```
p(f)
```

```
    1/8*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] - 1/8*p[2, 2] - 1/4*p[4]
```

```
s[2].plethysm(s[1,1])
```

```
    s[1, 1, 1, 1] + s[2, 2]
```

```
## This code was written by Jair Taylor to find the chromatic symmetric function of a graph.  In particular, all
of the chromatic symmetric functions for trees on up to 8 vertices are shown.

def ChromSym(V,E):
    #Finds the chromatic symmetric function of the (hyper)graph (V,E) by brute force.
    Sym = SymmetricFunctions(SR)
    mon = Sym.monomial()
    X = 0
    n = len(V)
    for P in Partitions(n):

        C = []
        for i in range(len(P)):
            C+= [i]*P[i]
        a = 0
        for chi in Permutations(C):
            chidict = {}
            for i in range(n):
                chidict[V[i]] = chi[i]
            for S in E:
                if len(set([chidict[v] for v in S])) == 1:
                    break
            else:
                a += 1
        X += a * mon(P)
    return X
```

```
V = [1,2,3,4,5]

E = [[1,2],[2,3],[3,4],[4,5], [5,3],[3,1]]

ChromSym(V,E)   #should return 120*m[1, 1, 1, 1, 1] + 24*m[2, 1, 1, 1] + 4*m[2, 2, 1]
```

```
    120*m[1, 1, 1, 1, 1] + 24*m[2, 1, 1, 1] + 4*m[2, 2, 1]
```

```
from sage.graphs.trees import TreeIterator
```

```
def CountTrees(n):
    count = 0
    for t in TreeIterator(n):
        count += 1
    return count
```

```
#n=15 7741
CountTrees(15)
```

```
    7741
```

```
CountTrees(5)
```

```
    3
```

```
def ConvertEdges(G):
    L = []
    for a in G.edges():
        L.append([a[i] for i in [0,1]])
    return L
def ChromSymTrees(n):
    L = []
    for t in TreeIterator(n):
        print t.degree_sequence()
        print ChromSym(t.vertices(),ConvertEdges(t))
    return L
```

```
ChromSymTrees(5)
```

```
    [2, 2, 2, 1, 1]
    120*m[1, 1, 1, 1, 1] + 36*m[2, 1, 1, 1] + 12*m[2, 2, 1] + 2*m[3, 1, 1] +
    m[3, 2]
    [3, 2, 1, 1, 1]
    120*m[1, 1, 1, 1, 1] + 36*m[2, 1, 1, 1] + 10*m[2, 2, 1] + 4*m[3, 1, 1] +
    m[3, 2]
    [4, 1, 1, 1, 1]
    120*m[1, 1, 1, 1, 1] + 36*m[2, 1, 1, 1] + 6*m[2, 2, 1] + 8*m[3, 1, 1] +
    m[4, 1]
    []
```

Quasisymmetric Functions

A quasisymmetric function is a power series in countably many variables which is invariant under shifts of the variables. The quasisymmetric functions form a ring under usual addition and multiplication of power series. There are several important bases for this ring. Here is a brief start computing quasisymmetric function expansions.

```
QSym=QuasiSymmetricFunctions(QQ)
```

```
QSym.inject_shorthands()
```

```
    Injecting M as shorthand for Quasisymmetric functions over the Rational
    Field in the Monomial basis
    /Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-\
    packages/sage/categories/sets_cat.py:1228: RuntimeWarning: redefining
    global value `M`
      inject_variable(shorthand, realization)
    Injecting F as shorthand for Quasisymmetric functions over the Rational
    Field in the Fundamental basis
    /Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-\
    packages/sage/categories/sets_cat.py:1228: RuntimeWarning: redefining
    global value `F`
      inject_variable(shorthand, realization)
    Injecting dI as shorthand for Quasisymmetric functions over the Rational
    Field in the dualImmaculate basis
    /Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-\
    packages/sage/categories/sets_cat.py:1228: RuntimeWarning: redefining
    global value `dI`
      inject_variable(shorthand, realization)
```

```
M(F[4,1,1])
```

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

```
F(s[3,1])
```

```
    F[1, 3] + F[2, 2] + F[3, 1]
```

```
#QSym?
```

Stanley Symmetric Functions

For each permutation there exists a Stanley symmetric function.

```
G = WeylGroup(['A',3])
```

```
w = G.from_reduced_word([3,2,3,1]) # [4,1,3,2] in one-line notation
```

```
w
```

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

```
w.reduced_words()
```

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

```
f=w.stanley_symmetric_function()
```

```
F(f)
```

```
    F[1, 3] + F[2, 2] + F[3, 1]
```

```
s(f)
```

```
    s[3, 1]
```

```
f
```

```
    3*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2] + m[3, 1]
```

Generating Functions

A generating function is a power series where the coefficients have combinatorial significance.   We can use algebraic manipulations on generating functions in proofs.   Sometimes we can also get nice closed forms.   Here we show how to use Sage to get the nth coefficient of a generating function.

```
var('x')  # allows us to do symbolic computation with x
```

```
    x
```

```
expand((1+x)*(1+x^2)*(1+x^3))
```

```
    x^6 + x^5 + x^4 + 2*x^3 + x^2 + x + 1
```

```
P = expand(prod((1+x^k) for k in range(1,10)))
P
```

```
x^45 + x^44 + x^43 + 2*x^42 + 2*x^41 + 3*x^40 + 4*x^39 + 5*x^38 + 6*x^37
+ 8*x^36 + 9*x^35 + 10*x^34 + 12*x^33 + 13*x^32 + 15*x^31 + 17*x^30 +
18*x^29 + 19*x^28 + 21*x^27 + 21*x^26 + 22*x^25 + 23*x^24 + 23*x^23 +
23*x^22 + 23*x^21 + 22*x^20 + 21*x^19 + 21*x^18 + 19*x^17 + 18*x^16 +
17*x^15 + 15*x^14 + 13*x^13 + 12*x^12 + 10*x^11 + 9*x^10 + 8*x^9 + 6*x^8
+ 5*x^7 + 4*x^6 + 3*x^5 + 2*x^4 + 2*x^3 + x^2 + x + 1
```

```
P.coeff(x^10)
```

```
9
```

```
[P.coeff(x^k) for k in range (0,10)] # compare to the number of partitions of n into distinct parts
```

```
[0, 1, 1, 2, 2, 3, 4, 5, 6, 8]
```

```
Q = expand(prod(1/(1-x^k) for k in range(1,10)))
```

```
Q.coeff(x^5)
```

```
0
```

```
Q
```

```
-1/((x - 1)*(x^2 - 1)*(x^3 - 1)*(x^4 - 1)*(x^5 - 1)*(x^6 - 1)*(x^7 -
1)*(x^8 - 1)*(x^9 - 1))
```

```
expand?
```

File: /Applications/sage.app/Contents/Resources/sage/local/lib/python2.7/site-packages/sage/calculus/functional.py

Type: <type ‘function’>

Definition: expand(x, *args, **kwds)

Docstring:

EXAMPLES:

```
sage: a = (x-1)*(x^2 - 1); a
(x - 1)*(x^2 - 1)
sage: expand(a)
x^3 - x^2 - x + 1
```

You can also use expand on polynomial, integer, and other factorizations:

```
sage: x = polygen(ZZ)
sage: F = factor(x^12 - 1); F
(x - 1) * (x + 1) * (x^2 - x + 1) * (x^2 + 1) * (x^2 + x + 1) * (x^4 - x^2 + 1)
sage: expand(F)
x^12 - 1
sage: F.expand()
x^12 - 1
sage: F = factor(2007); F
3^2 * 223
sage: expand(F)
2007
```

Note: If you want to compute the expanded form of a polynomial arithmetic operation quickly and the coefficients of the polynomial all lie in some ring, e.g., the integers, it is vastly faster to create a polynomial ring and do the arithmetic there.

```
sage: x = polygen(ZZ)      # polynomial over a given base ring.
sage: f = sum(x^n for n in range(5))
sage: f*f                  # much faster, even if the degree is huge
x^8 + 2*x^7 + 3*x^6 + 4*x^5 + 5*x^4 + 4*x^3 + 3*x^2 + 2*x + 1
```

TESTS:

```
sage: t1 = (sqrt(3)-3)*(sqrt(3)+1)/6;
sage: tt1 = -1/sqrt(3);
sage: t2 = sqrt(3)/6;
sage: float(t1)
-0.577350269189625...
sage: float(tt1)
-0.577350269189625...
sage: float(t2)
```

```
0.28867513459481287
sage: float(expand(t1 + t2))
-0.288675134594812...
sage: float(expand(tt1 + t2))
-0.288675134594812...
```

Matrices

Matrices are well supported in Sage. Here we calculate a Schur function on 3 variables as a ratio of determinants.

```
R = PolynomialRing(QQ, 9, 'x')   #allows symbolic computation with 9 variables
A = matrix(R, 3, 3, R.gens()); A
```

```
    [x0 x1 x2]
    [x3 x4 x5]
    [x6 x7 x8]
```

```
M= Matrix (R,3,3);M
```

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

```
for i in [0,1,2]:
    for j in [0,1,2]:
        M[i,j]=R.gens()[i]^j
M
```

```
    [   1   x0 x0^2]
    [   1   x1 x1^2]
    [   1   x2 x2^2]
```

```
M.det()
```

```
    -x0^2*x1 + x0*x1^2 + x0^2*x2 - x1^2*x2 - x0*x2^2 + x1*x2^2
```

```
def foo(a):
    MM= Matrix (R,len(a),len(a))
    for i in range(len(a)):
        for j in range(len(a)):
            MM[i,j]=R.gens()[i]^(a[j])
    return(MM)
```

```
D=foo([2,1,0]).determinant()
```

```
D
```

```
    x0^2*x1 - x0*x1^2 - x0^2*x2 + x1^2*x2 + x0*x2^2 - x1*x2^2
```

```
B=foo([5,2,0]).determinant(); B
```

```
    x0^5*x1^2 - x0^2*x1^5 - x0^5*x2^2 + x1^5*x2^2 + x0^2*x2^5 - x1^2*x2^5
```

```
B/D
```

```
    x0^3*x1 + x0^2*x1^2 + x0*x1^3 + x0^3*x2 + 2*x0^2*x1*x2 + 2*x0*x1^2*x2 +
    x1^3*x2 + x0^2*x2^2 + 2*x0*x1*x2^2 + x1^2*x2^2 + x0*x2^3 + x1*x2^3
```

```
#B/D to the Schur function s[3,1] on variables x1,x2,x3
s[3,1].expand(3)
```

```
    x0^3*x1 + x0^2*x1^2 + x0*x1^3 + x0^3*x2 + 2*x0^2*x1*x2 + 2*x0*x1^2*x2 +
    x1^3*x2 + x0^2*x2^2 + 2*x0*x1*x2^2 + x1^2*x2^2 + x0*x2^3 + x1*x2^3
```

```
B/D == s[3,1].expand(3)
```

```
True
```