Group Theory and SAGE: A Primer

Robert A. Beezer
University of Puget Sound

Version 1.1
March 3, 2009

Introduction

This `compilation` collects SAGE commands that are useful for a student in an introductory course in 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 `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, the "E" in SAGE once stood for "Exploration."

This guide is 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.

Basic Properties of the Integers

Integer Division

`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,

```
r = 14 % 3
q = (14 - r)/3
r, q
```

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,

```
gcd(2776, 2452)
```

will return 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.)

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

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.

```
answer1 = ((20 % 5) == 0)
answer2 = ((17 % 4) == 0)
answer1, answer2
```

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,

```
factor(2600)
```

returns `2^3 * 5^2 * 13`. We can strip off 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.

```
n = 2600
decomposition = factor(n)
print n, " decomposes as ", decomposition
```

```
secondterm = decomposition[1]
print "Base and exponent (pair) for second prime: ", secondterm
base = secondterm[0]
exponent = secondterm[1]
print "Base is ", base
print "Exponent is ", exponent
thirdbase = decomposition[2][0]
thirdexponent = decomposition[2][1]
print "Base of third term is ", thirdbase, " with exponent ", thirdexponent
```

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

Multiplicative Inverse, Modular Arithmetic

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

```
inverse_mod(352, 917)
```

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

Then try

```
inverse_mod(4, 24)
```

and explain the result.

Powers with Modular Arithmetic

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

```
power_mod(15, 831, 23)
```

returns `10`. If $m = -1$, then this command will duplicate the function of `inverse_mod()`.

Euler ϕ-function

`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,

```
euler_phi(345)
```

should return 176. Experiment by running the following code several times:

```
m = random_prime(10000)
n = random_prime(10000)
m, n, euler_phi(m*n) == euler_phi(m)*euler_phi(n)
```

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,

```
is_prime(117371)
```

returns `True`, while

```
is_prime(14547073)
```

returns `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

```
random_prime(10^21,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,

```
prime_range(500,550)
```

returns `[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 'em 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

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

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,

```
sigma = G("(1,3)(2,5,4)")
sigma.order()
```

will return 6.

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

```
sigma = G("(1,3)(2,5,4)")
sigma.sign()
```

will return -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'll get a list of available functions (you may need to scroll down to see the whole list). Experiment and explore! It's what SAGE is all about. You really can't 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/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

```
H = DihedralGroup(6)
```

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'll get a list of available functions (you may need to scroll down to see the whole list). As before, *experiment and explore* — its really hard to break anything.

Here's another couple of ways to experiment and explore. Find a function that looks interesting, say `is_abelian()`. Type `H.is_abelian(` (note there is just an opening parenthesis), followed by the tab 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 won't keep repeating its definition below.

Abelian?

The command

```
H.is_abelian()
```

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

Order

The command

```
H.order()
```

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

All Elements

The command

```
H.list()
```

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.

```
elements = H.list()
elements[3]
```

Cayley Table

The command

```
H.cayley_table()
```

will construct the Cayley table (or "multiplication table") of H. The display uses the elements of the group in the same order as the `list()` command, and denoting them as `xN` where N is an integer. For example to determine the element in the table named `x4`,

```
H.list()[4]
```

Center

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

```
H.center().list()
```

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 $\langle (1\,2\,3)(4\,5)\rangle = H \subseteq G = S_5$.

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

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.

```
n=20
CN = CyclicPermutationGroup(n)
for g in CN:
  print g.order(), "  ", g
```

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,

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

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

```
C20.conjugacy_classes_subgroups()
```

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.

```
K=DihedralGroup(12)
sg = K.conjugacy_classes_subgroups()
print "sg:\n", sg
print "\nAn order two subgroup:\n", sg[1].list()
```

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).

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

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) will have $2n$ 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 tetahedron as 1, 2, 3 and 4. Fix the vertex labeled 4 and roate 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,

```
tetra_one = PermutationGroup(["(1,2,3)", "(2,3,4)"])
tetra_two = PermutationGroup(["(1,2,3)", "(1,2)(3,4)"])
```

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

```
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)"])
cube.list()
```

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,

```
cube.is_isomorphic(SymmetricGroup(4))
```

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,

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

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

```
cubeface.is_isomorphic(SymmetricGroup(4))
```

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`.

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

Quotient Group

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

```
A4.quotient_group(H)
```

will return 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.

```
print AlternatingGroup(5).is_simple()
print AlternatingGroup(4).is_simple()
```

will print `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 conjugagcy 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,

```
G = SymmetricGroup(5)
group_order = G.order()
reps = G.conjugacy_classes_representatives()
class_sizes = []
for g in reps:
  class_sizes.append( group_order/G.centralizer(g).order() )
print class_sizes
```

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 8 symbols, S_8, has order $8! = 40,320$ and is divisible by $2^7 = 128$. Let's find one example of a subgroup of permutations on 8 symbols with order 128. The next command takes a few minutes to run, so go get a cup of coffee after you set it in motion.

```
G = SymmetricGroup(8)
subgroups = G.conjugacy_classes_subgroups()
map( order, subgroups )
```

The `map(order, subgroups)` command will apply the `order()` method to each of the subgroups in the list `subgroups`. The output is thus a large list of the orders of many subgroups (296 to be precise). if you count carefully, you will see that 259^{th} subgroup has order 128. You can retrieve this group for further study by referencing it as `subgroups[258]` (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	`PermutationGroup(["(1,2,5,6)(3,4,7,8)",` `"(1,3,5,7)(2,8,6,4)" ])`	Quaternions The two generators are I and J
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	`PermutationGroup(["(1,2,3)(4,6)(5,7)",` `"(1,2)(4,5,6,7)"])`	Non-abelian 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!