Toward the Code - Chennai

Moving Toward the Code

This Sage worksheet is for the course in Sage and programming at the Institute of Mathematical Sciences in Chennai, circling around Sage Days 60.

We are now ready to start going in deeper toward the code of Sage. In the near future, the course will cover object-oriented programming and classes which is a way to structure your program so that, as we've said before, you always have the right things available to the right mathematical objects. But in order to do that, we need to see more ways to interact with the code, and how to follow it.

Before that, let's see any interacts you made, as a preparation! Here's one I cooked up for fun.

```
@interact
def _(order=4):
    G = DihedralGroup(order)
    H = G.subgroups()
    html("There are $%s$ subgroups of $D_{%s}$, of orders "%
(len(H),order)+str([h.order() for h in H]))
```

The Sage Codebase

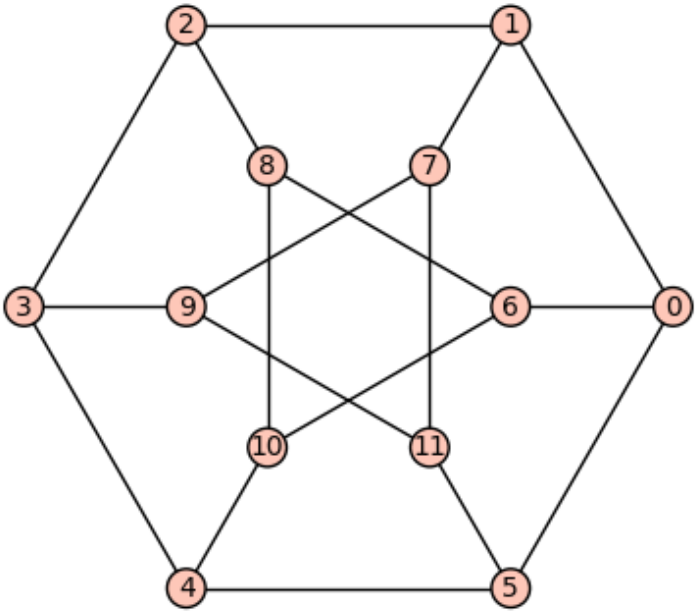
The goal of this course is not just to get you to be able to *use* Sage, but to write your own code using it as well. In order to do this effectively, finding out how to search the Sage codebase is very useful. This is especially true if your work is useful and could be adopted inside of Sage itself.

Unfortunately for beginners, the Sage codebase is very, very large - lakhs and lakhs of lines of code specifically written for Sage, especially in areas like combinatorics and number theory, and then similar amounts "wrapping" pre-existing work from other open-source projects for a unified interface. Here are some examples of each.

Graph Theory

The Dürer graph is implemented directly in Sage.

```
G = graphs.DurerGraph()
show(G)
```



We can see the code here, or at Sage's repository of code.

```
graphs.DurerGraph??
```

This code seems pretty straightforward; we define a dictionary of edge relations "edge_dict", a dictionary for positioning a nice picture "pos_dict", and we feed them into "Graph" and are done. Only... what was the "Graph" thing?

```
Graph??
```

Whew! That is a little overwhelming! Let's break it down.

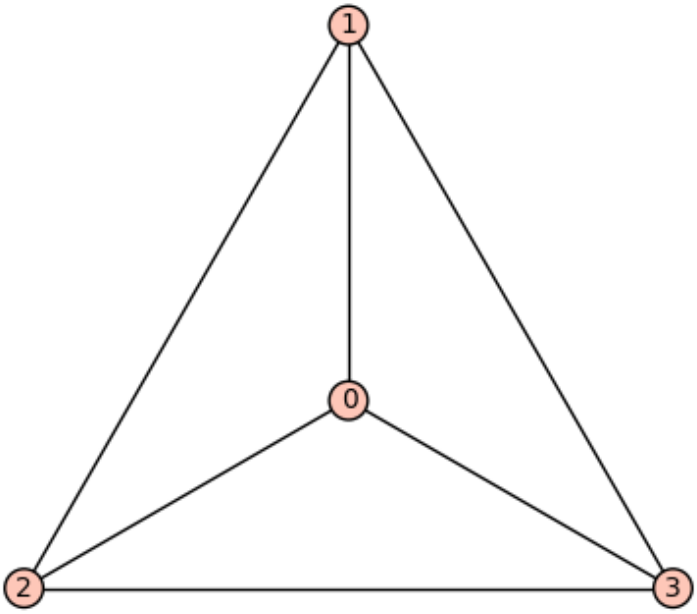
- First, there is the "class". You will learn what this actually is later; for now, just think of it as a way to make lots of different graphs but with the same notation.
- Then there is a lot of documentation, including examples that are automatically tested if one runs tests in Sage.
- Then comes "__init__", a method which has some defaults but which takes as arguments things that look familiar like position.
- Finally it runs through a very large amount of code to figure out, for any way to input a graph, exactly which one the user has asked for and (eventually) gives it back.

In particular, in this case we fed "__init__" a dictionary of edge relations and fed it "pos=pos_dict", which specifies the layout when displaying the graph. That is enough for it to know everything to make the graph.

Since covering this more is really the beginnings of object-oriented programming, I will tread lightly here. But I encourage you to start reading through it even now.

Here is a different kind of example.

```
H = graphs.TetrahedralGraph()
show(H)
```




```
graphs.TetrahedralGraph??
```

You might ask what's so different about this! But it is quite different. See these two lines.

```
import networkx
G = networkx.tetrahedral_graph()
```

As you can see, the tetrahedral graph is not constructed the same way. Although in the next line it still calls the "Graph" *constructor* with a position dictionary, Sage has no longer created this from scratch. What happened?

It turns out that Los Alamos National Laboratory in New Mexico, USA helped start work on an open-source Python library called NetworkX for doing network analysis - like graph theory. And it is a well-developed and good set of tools, so why reinvent the wheel? Sage comes with NetworkX included, so we just "import" it (something you'll learn about in the OO part of the course) and then just use it like any other Python function. Hooray!

Groups

Most groups are not implemented natively, but with a blend of Sage and GAP. How this happens is instructive in how Sage tries to keep things mathematically organized as well as use the best solutions.

Let's create a group first.

```
P = PermutationGroup([[(1,2,3)],[(2,3,4)]])
```

```
type(P)
```

```
    <class
    'sage.groups.perm_gps.permgroup.PermutationGroup_generic_with_catego\
    ry'>
```

Note that at this time we have not yet used GAP, to save time and energy. At the command line, I can verify this:

```
Last login: Tue Aug 12 02:12:28 on ttys005
dhcp79:~ karl.crisman$ sage
```

```
Sage Version 6.2, Release Date: 2014-05-06
Type "notebook()" for the browser-based notebook interface.
Type "help()" for help.
```

```
sage: P = PermutationGroup([[(1,2,3)],[(2,3,4)]])
sage:
Exiting Sage (CPU time 0m0.07s, Wall time 0m8.41s).
dhcp79:~ karl.crisman$
```

Some methods will not need to use that computation power, either.

```
P.gens()
```

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

However, most nontrivial computations will require Sage to start up GAP internally; these are *wrappers* of GAP functionality (sometimes quite sophisticated ones).

```
P.center()
```

```
Subgroup of (Permutation Group with generators [(2,3,4), (1,2,3)])
generated by [()]
```

Let's verify it at the command line:

```
sage: P.center()
Subgroup of (Permutation Group with generators [(2,3,4), (1,2,3)]) generated by
[()]
sage:
Exiting Sage (CPU time 0m0.10s, Wall time 0m6.98s).
Exiting spawned Gap process.
```

Okay, now let's leave the notebook and dive into following how Sage creates a group and keeps it in order. Warning: you should *not* expect to follow this the first time! But it is only by being exposed to the full structure that you will start seeing where your ideas can fit in.

```
G = AlternatingGroup([1,2,4,5])  # from the documentation
```

One handy way to look for things is with "search_src". (For finding functions and methods, "search_def" is likewise useful.)

```
search_src("AlternatingGroup")
```

Search Source Code: "AlternatingGroup"

1. categories/finite_groups.py
2. categories/finite_permutation_groups.py
3. categories/groups.py
4. categories/homset.py
5. categories/magmas.py
6. categories/semigroups.py
7. geometry/polyhedron/library.py
8. graphs/generic_graph.py
9. groups/class_function.py
10. groups/finitely_presented_named.py
11. groups/group.pyx
12. groups/groups_catalog.py
13. groups/old.pyx
14. groups/perm_gps/all.py
15. groups/perm_gps/permgroup.py
16. groups/perm_gps/permgroup_element.pyx
17. groups/perm_gps/permgroup_named.py
18. groups/perm_gps/permutation_groups_catalog.py
19. homology/examples.py
20. homology/simplicial_complex.py
21. interfaces/interface.py
22. libs/gap/gap_functions.py
23. libs/gap/libgap.pyx
24. matrix/operation_table.py
25. rings/number_field/number_field.py
26. structure/parent.pyx
27. tests/parigp.py

It turns out that src/groups/perm_gps/permgroup_named.py is where we want to look first. This corresponds to http://git.sagemath.org/sage.git/tree/src/sage/groups/perm_gps/permgroup_named.py on git.sagemath.org and http://www.sagemath.org/doc/reference/groups/sage/groups/perm_gps/permgroup_named.html in the Sage documentation. We get:

```
PermutationGroup_symalt.__init__(self, gap_group='AlternatingGroup(%s)'%len(domain), domain=domain)
```

```
PermutationGroup_
```

Hmm, I don't see this. Where could it be? Let's look at the top of the file for it... oh, it's "import"ed. (And here we begin the chase down the rabbit hole. Be careful to note the "domain" and where it goes.)

```

```

```

```

(After more digging...)

Yikes! But this is how to start going through things. Granted, this is more complex than most. But the key is that we

separate Sage from its subsystems. As soon as we need GAP functionality, we just (internally) do

```
G._gap_()
```

and that enables us access to GAP methods. Is there one we didn't already wrap? No problem:

```
G1 = G._gap_()
```

```
G1.ComputedPCentralSeriess()
```

```
[  ]
```

So you see that we even have access to more than we thought!

Combining things

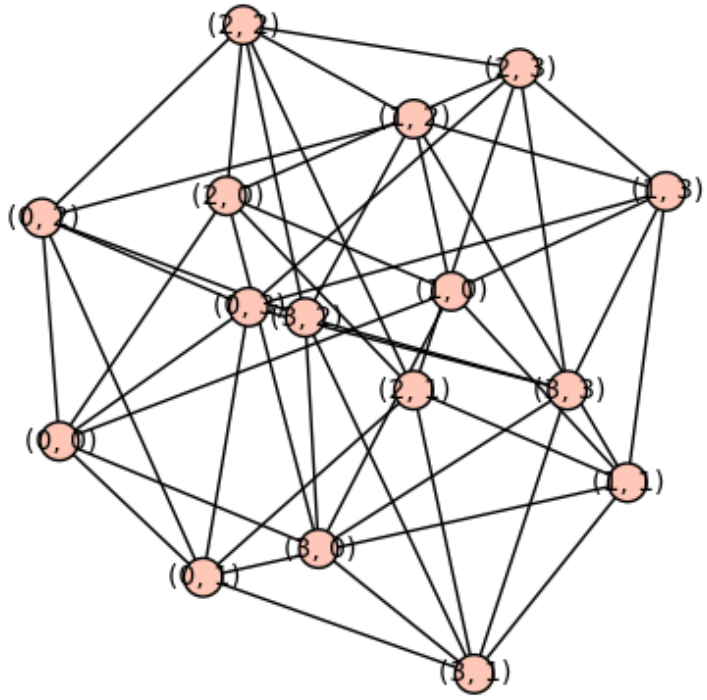
The beauty of this is that we can combine things that otherwise are challenging to combine. The following example speaks for itself, I think, especially if we look at the code for each command in turn.

```
H # Takes NetworkX graph
```

```
Tetrahedron: Graph on 4 vertices
```

```
H = H.cartesian_product(H) # Uses Sage functionality
```

```
show(H) # uses matplotlib
```



```
Sym = H.automorphism_group() # Native Sage Code using C backend for graphs
```

```
len( Sym.list() )  # Needs GAP to calculate
```

```
1152
```

```
Sym._gap_().ComputedPCentralSeriess()  # Directly within GAP
```

```
[  ]
```

Errors

In principle, one can follow errors this way too. It can be harder, though, if "compiled" components are involved.

```
H = H.cartesian_product(H)
Sym = H.automorphism_group()
len( Sym.list() )
```

```
Traceback (click to the left of this block for traceback)
...
   executing Elements($sage1);
```

```
Sym.joke()
```

```
Traceback (click to the left of this block for traceback)
...
AttributeError: 'PermutationGroup_generic_with_category' object has
no attribute 'joke'
```

Keeping track

This can seem completely hopeless. How on earth do we keep track of all of this code?

There are two key elements. The first should be obvious by now: the Sage source is organized by different types in a large tree of functionality. This is roughly paralleled in the reference manual as well.

However, the second element is something that most mathematicians (and indeed, most people) are not aware of. That missing piece is called *revision control*. In order to introduce it, let's tell a story.

- Imagine you are writing a paper. You start by writing some boilerplate and then dive in.
- Halfway through you realize you really need a collaborator and add her.
- You send emails back and forth with the .tex file attached.
- After a while you realize that you and the collaborator actually are writing two different papers that could be separated.
- But - oh, oh, now you are so entangled you don't remember who wrote what! So you both put both your names on both papers.

Okay, that is a little corny. But my point is that it does matter how you keep track of who did what and when. This is important for many reasons, whose importance will depend on the context. All of these apply to writing code with more than one person.

- Assigning credit for contributions
- Assigning blame for contributions...
- Determining exactly when a particular bug crept in
- Determining exactly when a bug was fixed
- Allowing people to work independently and then bring their changes together
- Preserving a record of this history even if the project leader quits
- Your reasons?

Sage has a fairly complex system for keeping track of proposing changes and keeping track of changes. You already know about the code, so let's give a brief introduction to the next steps.

Trac

Sage has a central place for all discussions about proposed new code, desired functionality, or bugfixes, called the Trac server, http://trac.sagemath.org/ . There are many different ways people use this.

- Report desired new behavior/enhancement, such as http://trac.sagemath.org/ticket/16799 .
  - Note how there is a component for easy searching, one can cc: developers you know will be interested, etc.
  - Discussions about the best way to implement things are often verrrrrrry long...
- Report bugs and discuss how to solve them.
  - As an example, http://trac.sagemath.org/ticket/16796 has one person reporting, another providing a fix, and a third one testing. This is very common.
  - You can search *all* bugs by component, even: http://trac.sagemath.org/report/64
- http://trac.sagemath.org/wiki/SageCombinatRoadMapis an example of a wiki page where a group of developers track things which need to be done in a particular component.

It is not too hard to get a new account on Trac, but the point here is just to show you what is possible and is going on daily in Sage development.

One particularly important thing is how to see proposed *changes* in the code. As an example, the OS X problem ticket above looks like this at the top.

The green link under "Branch" (in this case it's green because it is based on the current version of Sage) brings us to here. It shows all changes and additions and deletions from the "current" code to the proposed changes.

Building Sage

So how do we start interacting with this system instead of writing our own worksheets? There are three things we need to start.

1. A command-line interface as I have occasionally demonstrated. This can be on Linux or Mac.
2. The Sage installation guide at http://sagemath.org/doc/installation/
3. The prerequisites for "compiling" Sage - that is, making Sage work from scratch. This is easy on Linux, a little more annoying on Mac.

We will not go through this in detail. Suffice to say that the steps are good but you should not hesitate to ask one of the help lists if you have trouble.

Revision Control

Once you have made a brand-new Sage, you are now ready to start making changes. Let's go to the terminal and see this "live".

(Demo)

What we have just seen is *distributed revision control*. We are keeping track of our changes, but there is not necessarily any central place all changes go (though in practice there is with Sage).

The Sage developer guide http://sagemath.org/doc/developer/ has lots (and lots) of information about how to use this effectively. Unfortunately it requires some nontrivial configuration and a somewhat steep learning curve, and we will save that for another lecture.

The point is that with a system integrating like this, we can keep track of all the many changes people propose to Sage, how they fit together, where there are conflicts, and integrate this with the process of *review* - similar to academic peer-review - to make Sage a better system. And you are ready to start it!

Next up: Sage Days!

You are now ready to participate very fully in Sage Days 60. Good luck and also enjoy the remainder of this course in Sage, math, and programming! Thank you for the opportunity.