

Programming Tutorial 2 Chennai

Sage Advanced Programming Tutorial

This [Sage](#) worksheet is for the course in Sage and programming at the [Institute of Mathematical Sciences](#) in Chennai, circling around [Sage Days 60](#). It is based on tutorials developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071).

There are far more useful programming tips and techniques available than we could cover in weeks. Nonetheless, certain structures and concerns are particularly important as you start to do more serious mathematics with Sage. Hopefully some of the homework from last time convinced you of the need for more tools!

This tutorial aims to cover some more slightly more advanced aspects of lists and functions, as well as some new datatypes and other things to be aware of. Think of it as a potpourri of possibly useful ideas, not a comprehensive list.

Structure and Flow

We saw the "block" structure of Python in the last tutorial, with the all-important indentation.

```
G = groups.permutation.Quaternion()  
L = []  
for g in G.subgroups():  
    L.append(g)
```

L

```
[Permutation Group with generators [()], Permutation Group with  
generators [(1,3)(2,4)(5,7)(6,8)], Permutation Group with generators  
[(1,3)(2,4)(5,7)(6,8), (1,5,3,7)(2,8,4,6)], Permutation Group with  
generators [(1,2,3,4)(5,6,7,8), (1,3)(2,4)(5,7)(6,8)], Permutation  
Group with generators [(1,3)(2,4)(5,7)(6,8), (1,6,3,8)(2,5,4,7)],  
Permutation Group with generators [(1,2,3,4)(5,6,7,8),  
(1,3)(2,4)(5,7)(6,8), (1,5,3,7)(2,8,4,6)]]
```

This created a list of all subgroups of the quaternion group.

What happens if I want to create something more interesting, though? For instance, I might want a list of a couple random elements of each subgroup of a particular group (so, possibly duplicating some elements).

```
L = [] # remember to reset L!
for g in G.subgroups():
    for i in range(2):
        L.append( g.random_element() )
```

L

```
[(), (), (1,3)(2,4)(5,7)(6,8), (), (1,7,3,5)(2,6,4,8), (), (),
(1,2,3,4)(5,6,7,8), (1,6,3,8)(2,5,4,7), (1,6,3,8)(2,5,4,7),
(1,8,3,6)(2,7,4,5), (1,6,3,8)(2,5,4,7)]
```

Notice the nested indentation and colons. Any text which comes after a "#" symbol (technically known as octothorpe) is a *comment*, ignored completely by Sage.

It can actually be harder to create a list of lists. I recommend judicious combinations of list comprehensions and loops. Look at the following examples listing various elements and other information about subgroups of this group.

```
L = []
for g in G.subgroups():
    L.append(g.list())
```

L

```
[[()], [(1,3)(2,4)(5,7)(6,8)], [(1,3)(2,4)(5,7)(6,8),
(1,5,3,7)(2,8,4,6), (1,7,3,5)(2,6,4,8)], [(1,2,3,4)(5,6,7,8),
(1,3)(2,4)(5,7)(6,8), (1,4,3,2)(5,8,7,6)], [(),
(1,3)(2,4)(5,7)(6,8), (1,6,3,8)(2,5,4,7), (1,8,3,6)(2,7,4,5)], [(),
(1,2,3,4)(5,6,7,8), (1,3)(2,4)(5,7)(6,8), (1,4,3,2)(5,8,7,6),
(1,5,3,7)(2,8,4,6), (1,6,3,8)(2,5,4,7), (1,7,3,5)(2,6,4,8),
(1,8,3,6)(2,7,4,5)]]
```

```
L = []
for g in G.subgroups():
    L.append([ elt.order() for elt in g ])
```

L

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

True or False

Talking about block structure also gives the opportunity to introduce conditional statements and comparisons. For instance, how might we extract only elements of a group of a certain order? The quaternion group doesn't have any elements of order eight, so let's try with four.

```
for g in G: # the elements of the group
```

```
if g.order() == 4:
    print g
```

```
(1,2,3,4)(5,6,7,8)
(1,4,3,2)(5,8,7,6)
(1,5,3,7)(2,8,4,6)
(1,6,3,8)(2,5,4,7)
(1,7,3,5)(2,6,4,8)
(1,8,3,6)(2,7,4,5)
```

What just happened? Mathematically, six of the elements had order 4, and those are the ones printed by the loop. But what happened in terms of the computer?

Here we have not just a nested structure, but one where the second part is asking "if" a certain statement is true or not.

- If the statement after "if" is evaluated by Sage to be "True" (which means Sage can prove it is true), we do the next level of nested indentation.
- Otherwise, we move on.

It is **very important** to note that we used "g.order() == 4", and not a single equals sign. A single equals just means "give something this name"; a double equals means "see if these things are actually equal. It is a good idea, if you need to use comparisons, to familiarize yourself with [how Python compares elements](#). In a longer course on programming, one would spend significant time with Boolean operators and comparisons.

Here is another example that shows Sage does the mathematically sensible thing as often as possible. The fourth power of B is not the number zero, but the zero matrix, but this is a well-understood (non-)abuse of notation.

```
B = matrix([[0,1,0,0],[0,0,1,0],[0,0,0,1],[0,0,0,0]])
for i in range(5): # all integers from 0 to 4, remember
    if B^i == 0: # We ask if the power is the zero matrix
        print i
```

4

Even more blocks

There are lots more flow control tools that you can use. Here is an example that uses "if/else" and "try/except", two two-parter blocks that are very useful (as this example illustrates). In your homework you are asked to analyze this.

```
for i in range(10):
    try:
        F = factor(i)
        if len(F)==1:
            print "Just one factor for {}".format(i)
        else:
            print "Too bad, {} is composite".format(i)
    except ArithmeticError:
        print "You didn't try to factor zero, did you?"
```

```
You didn't try to factor zero, did you?
Too bad, 1 is composite
Just one factor for 2
Just one factor for 3
Just one factor for 4
Just one factor for 5
Too bad, 6 is composite
Just one factor for 7
Just one factor for 8
Just one factor for 9
```

Lists again (and their friends)

It's time to return to lists. Let's start by reviewing some things that were implied by the homework.

```
a=[1,2,3] # list
a
```

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

```
a[-1]
```

```
3
```

We can count backwards from the end. If we did "a[-2]" that would give us the second-to-last element (if one exists), and so forth.

More surprising, we can *change* elements of lists with this notation.

```
a[1] = -1
a
```

```
[1, -1, 3]
```

Unlike in some other programming languages, you can't extend lists this way.

```
a[3] = 2
```

```
Traceback (click to the left of this block for traceback)
```

```
...
```

```
IndexError: list assignment index out of range
```

Slicing lists

But that is just the tip. Python has a powerful facility for getting (and setting) multiple elements of lists in various ways, or "slicing" a list.

```
a=range(10)
a
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
a[1:4]
```

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

This notation essentially is short for "a[i] for $1 \leq i < 4$ ". So I can get a sublist very easily with this.

If the beginning value is not specified, it defaults to the first item. If the ending value is not specified, the slice goes to the end of the list.

```
a[:4] # everything up to, but not including element number 4
```

```
[0, 1, 2, 3]
```

```
a[4:]
```

```
[4, 5, 6, 7, 8, 9]
```

We can assign things to a list this way too.

```
a[1:4] = [2,3,4]
```

```
a
```

```
[0, 2, 3, 4, 4, 5, 6, 7, 8, 9]
```

However, without '=', these are copies of the list. In fact, it's a convenient way to copy a list!

```
a = range(10)
```

```
a[:] # a copy of the entire list
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

There are many variants on this syntax. See if you can figure out how to describe the behavior in the next few cells (this is your first homework problem). Think about what default behavior must be, among other things. The only way to internalize this is with lots of practice.

```
a[1:6:2]
```

```
[1, 3, 5]
```

```
a[::2]
```

```
[0, 2, 4, 6, 8]
```

```
a[::-1]
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
a[-3:]
```

```
[7, 8, 9]
```

```
a[3:-2:2]
```

```
[3, 5, 7]
```

Matrices slice too

Sage has adopted this slicing notation for matrices and vectors to allow powerful ways to get submatrices, reorder rows/columns, etc. Additionally, you can specify an explicit list of columns/rows for an index to construct submatrices. See the [documentation](#).

```
m=matrix(4,range(16))
m
```

```
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
[12 13 14 15]
```

```
m[::2,[0,3]] # GET every other row, columns 0 and 3
```

```
[ 0  3]
[ 8 11]
```

```
m[::2,[0,3]] = matrix([[ -1,2],[0,0]]) # SET every other row, columns 0
and 3
```

```
m # check it worked
```

```
[ -1  1  2  2]
[  4  5  6  7]
[  0  9 10  0]
[12 13 14 15]
```

```
m[::-1,:] # reverse the order of the rows
```

```
[12 13 14 15]
[  0  9 10  0]
[  4  5  6  7]
[ -1  1  2  2]
```

```
m[[-1,0],[1,2]] # GET the last row, then first, with columns 1 and 2
```

```
[13 14]
[ 1  2]
```

When slicing goes just one way

Recall the numerical integral question from homework. How do we get just the answer?

```
ANS = numerical_integral(x^3,0,1)
```

```
ANS; ANS[0]
```

```
(0.25, 2.7755575615628914e-15)
0.25
```

You might think we could now change "ANS" if it was convenient - after all, it's not the numerical integral, just a sequence of numbers. But you would be wrong.

```
ANS[0] = 4
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: 'tuple' object does not support item assignment
```

Sometimes, you want something like a list, but which is not in danger of being accidentally modified. In cases like this, you want a *tuple*.

```
b = (1,2,3) # tuple
b
(1, 2, 3)
```

We make tuples using commas without brackets. Typically, parentheses are also included for clarity, and I encourage this use.

Now let's try to change our tuple.

```
b[1] = -1 # gives error
Traceback (click to the left of this block for traceback)
...
TypeError: 'tuple' object does not support item assignment
```

There is a [lot you might want to do](#) with tuples. The most common thing, though, is to use them to return more than one thing from a function, and then to assign those outputs to variables. Like below.

```
(answer, error_bound) = numerical_integral(x^3,0,1)
print answer
print error_bound
0.25
2.77555756156e-15
```

Inherent in that is the use of tuples. For right now, the most important thing is to be aware that when you see an object like a list, but with *parentheses*, you can't change it - and that's a good thing.

Even more ways to construct lists

Remember the list comprehension "set-builder notation" way to make lists? This can include filtering tests as well.

```
[g for g in G if g.order() == 4]
[(1,2,3,4)(5,6,7,8), (1,4,3,2)(5,8,7,6), (1,5,3,7)(2,8,4,6),
(1,6,3,8)(2,5,4,7), (1,7,3,5)(2,6,4,8), (1,8,3,6)(2,7,4,5)]
```

The construction simply adds in the comparison statement. This can get quite complex. Treating these constructs as set builder notation is probably best.

```
[i+j for i in [1..10] for j in [1..10] if (i+j).is_prime() and i*j !=
10]
```

```
[2, 3, 5, 7, 5, 11, 7, 11, 13, 11, 13, 11, 13, 13, 17, 17, 19]
```

$$\{i+j | i, j \in \mathbb{Z}, 1 \leq i, j \leq 10, i+j \in \mathbb{P}, ij \neq 10\}$$

A list can also be constructed "lazily" using a *generator*. A generator generates its values as it is asked for them, not before. To make a generator, you use list comprehension notation, but with round parentheses.

```
(i for i in range(3))  
<generator object <genexpr> at 0x10eec71e0>
```

To actually enumerate all the values, you can use the "list()" function.

```
list(i for i in range(3))  
[0, 1, 2]
```

Instead of explicitly listing the values, you can do some other operation, like summing them.

```
sum(i for i in (1..10) if i.is_prime())  
17
```

This is not so important for basic Sage use, but for constructing more efficient code (so that you aren't wasting memory and time creating a whole list of what you want until you really need it) it is crucial

Sage has a huge number of generators that give interesting things. For example, here we make all graphs on 5 vertices.

```
graphs(5)  
<generator object __call__ at 0x10eec7280>
```

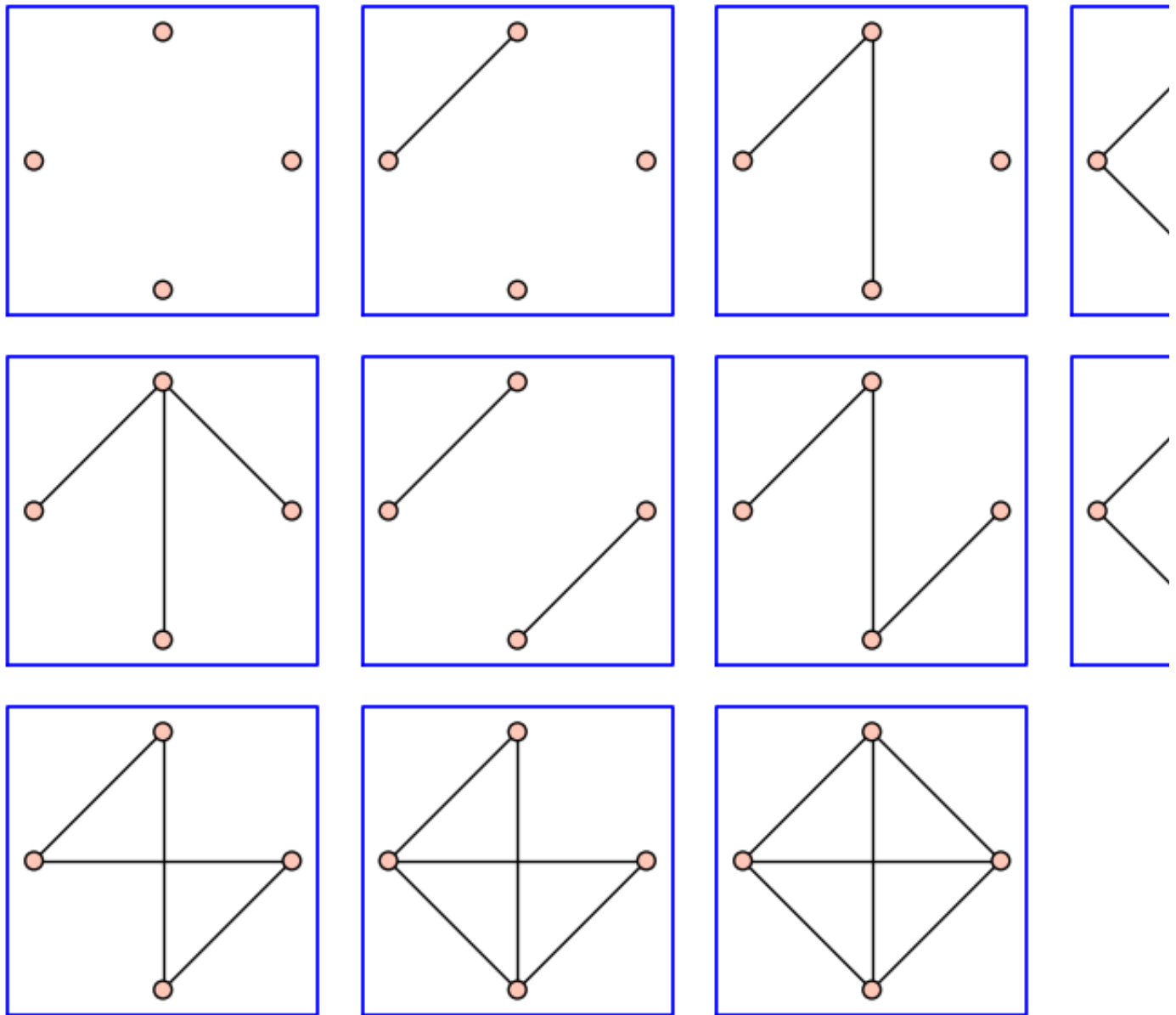
Let's time this and compare it with timing creating all of them.

```
%time  
G = graphs(5)  
CPU time: 0.00 s, Wall time: 0.00 s
```

```
%time  
len(list(G))  
34  
CPU time: 0.06 s, Wall time: 0.14 s
```

Generators are subtle beasts, though, so be careful:

```
G.next()  
Traceback (click to the left of this block for traceback)  
...  
StopIteration  
show(graphs(4))
```

Dictionaries

A fairly different, but crucial, datatype is a dictionary. What is a dictionary?

To show this, let's think of a common situation. Suppose you want to define a matrix that encodes some information about a combinatorial object. But ... nearly all the entries are zero. Do you *really* want to type in all those zeros?

```
matrix([[0,0,0,0],[0,1,0,0],[0,0,2,0],[0,0,0,-1]])
```

```
[ 0  0  0  0]
[ 0  1  0  0]
[ 0  0  2  0]
[ 0  0  0 -1]
```

Try this instead!

```
nonzero = { (1,1):1, (2,2):2, (3,3):-1 }
matrix(nonzero)
```

```
[ 0  0  0  0]
[ 0  1  0  0]
[ 0  0  2  0]
[ 0  0  0 -1]
```

Remember, the "(1,1)" spot is the *second* column and *second* row. What did I do to create this?

- I placed things inside curly braces.
- Each (nonzero) element of the matrix was separated by commas, just like in lists and tuples.
- I separated the entry location and the entry value with a colon each time.

Such an object is called a *dictionary*. This can be thought of as a mathematical mapping from "keys" to "values". (Here, the keys were the location and the values were the ... values.) There are a couple technical points you won't understand yet if you are new to them, but which the Python police require I mention:

- The order is *not* important and *not* guaranteed to hold; only the *relations* matter.
- The *keys* (the first element of a colon-pair) must be "hashable". So you can't have a list as a key.

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

```
Traceback (click to the left of this block for traceback)
```

```
...
```

```
TypeError: unhashable type: 'list'
```

```
{(1,2):3}
```

```
{(1, 2): 3}
```

We have to be careful; you can't add to the values easily.

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

```
{1: [3]}
```

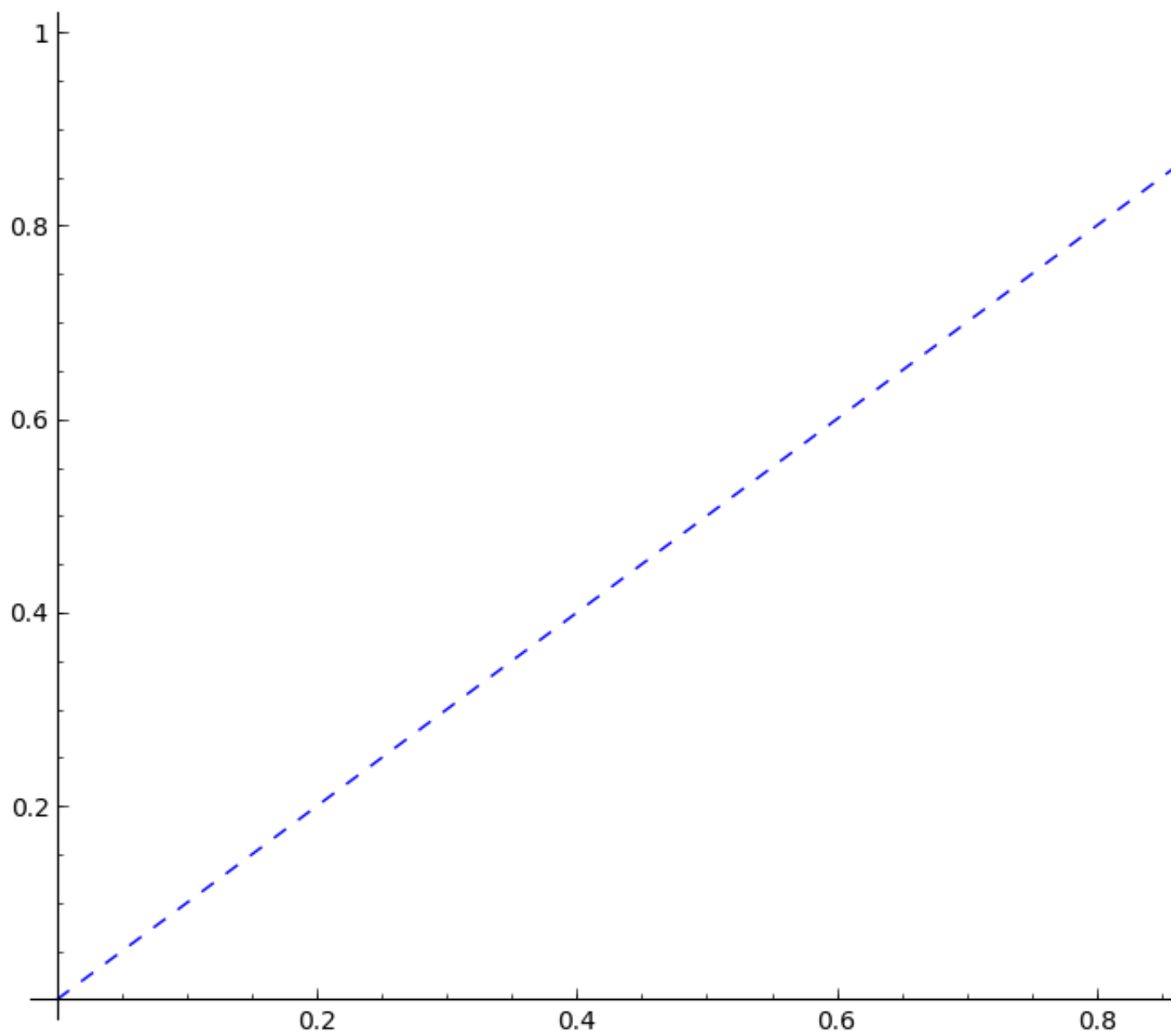
But in other ways, dictionaries are sort of like lists.

```
O = sage.plot.plot.plot.options; O
```

```
{'fillalpha': 0.5, 'detect_poles': False, 'plot_points': 200,
'thickness': 1, 'alpha': 1, 'adaptive_tolerance': 0.01, 'fillcolor':
'automatic', 'adaptive_recursion': 5, 'aspect_ratio': 'automatic',
'exclude': None, 'legend_label': None, 'rgbcolor': (0, 0, 1),
'fill': False}
```

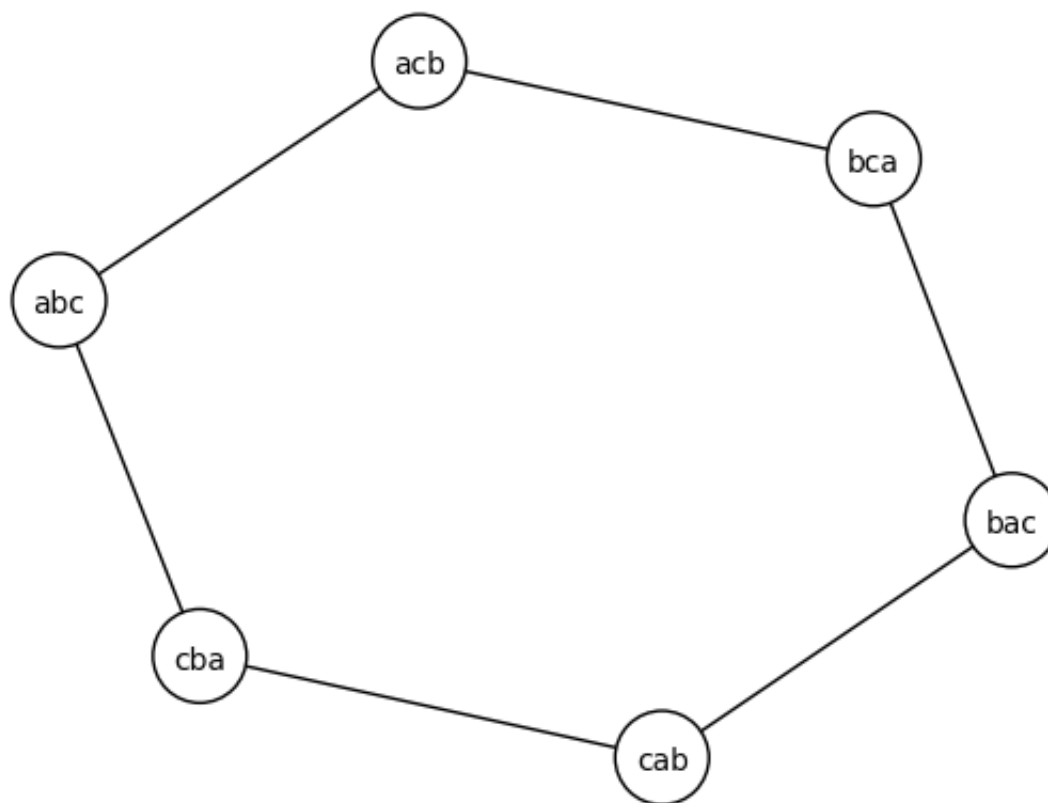
```
O['linestyle'] = '--'
```

```
plot(x,(x,0,1))
```



We can also use a generator to create a dictionary. This is very powerful for creating otherwise-tedious and error-prone mathematical objects.

```
str = 'abc'
elts = list(str)
LS1 = Permutations(elts).list()
LS2 = LS1[:]
d = dict( (''.join(list(ls1)),[''.join(list(ls2)) for ls2 in LS2 if
(ls1[0]==ls2[0] or ls1[1]==ls2[1]) and ls1 != ls2 ]) for ls1 in LS1 )
G = Graph(d)
P = plot(G,figsize=5,vertex_size=1000,vertex_colors='white')
P.show()
Gp = G.automorphism_group()
print Gp.order()
```



12

If I change to 'abcd' it now automatically creates the "right" graph for that case.

Two more topics

We'll end with two things that are worth being aware of where things aren't quite what they seem.

Lambda Functions

Sometimes you don't want to go to all the trouble of making a function (for instance, because it makes things more complex), but you nonetheless need a function. Lambda functions are short one-line functions similar to "def" functions which are very helpful in such situations.

- Technical note: lambda functions do not create a new local scope, while def functions do.

The syntax is very short. The input variables are before the colon, the output is after it.

```
f = lambda x,y: x+y
```

```
f(1,2)
```

3

Lest you think this is completely pointless (as the previous example was), here are some real-life examples of how they are used.

First, suppose you have a list of points like this one.

```
points=[(i,sin(RR(i))) for i in range(10)]
points
```

```
[(0, 0.0000000000000000), (1, 0.841470984807897), (2,
0.909297426825682), (3, 0.141120008059867), (4, -0.756802495307928),
(5, -0.958924274663138), (6, -0.279415498198926), (7,
0.656986598718789), (8, 0.989358246623382), (9, 0.412118485241757)]
```

Oops, but I want to sort them by the *dependent* variable. Now what?

No problem, just use a lambda function with the Python builtin "sorted".

```
sorted(points, key=lambda p: p[1])
```

```
[(5, -0.958924274663138), (4, -0.756802495307928), (6,
-0.279415498198926), (0, 0.0000000000000000), (3, 0.141120008059867),
(9, 0.412118485241757), (7, 0.656986598718789), (1,
0.841470984807897), (2, 0.909297426825682), (8, 0.989358246623382)]
```

The function "sorted" takes a function to sort by, and this function selects the second ("oneth") element of each coordinate pair.

To do this without a lambda is somewhat more tedious, and creates an unnecessary function that you might then accidentally use.

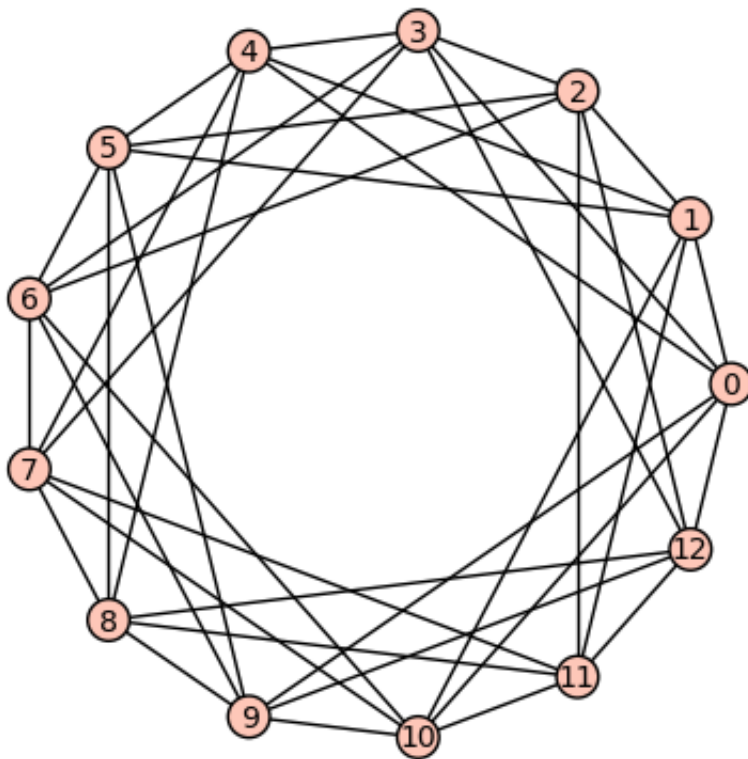
```
def find_second_element(p):
    return p[1]
sorted(points, key=find_second_element)
```

```
[(5, -0.958924274663138), (4, -0.756802495307928), (6,
-0.279415498198926), (0, 0.0000000000000000), (3, 0.141120008059867),
(9, 0.412118485241757), (7, 0.656986598718789), (1,
0.841470984807897), (2, 0.909297426825682), (8, 0.989358246623382)]
```

We can easily (well, at least with only one line of code) construct and display a [Paley graph](#) using this as well. Parse this carefully!

```
pos = dict([i,[cos(2*pi*i/13).n(),sin(2*pi*i/13).n()]] for i in
range(13))
g = Graph([GF(13), lambda i,j: i!=j and (i-j).is_square()], pos=pos)
```

```
show(g)
```



Annoying corollary

If for some reason you want to use "lambda" as a variable, you'll have to do this annoying thing.

```
var('lambda_')
lambda_^2-1
lambda_^2 - 1
```

This seems bad, but in this one case we have hacked Sage so that showing the expression still shows the Greek letter.

```
show(lambda_^2-1)
```

$\lambda^2 - 1$

Kinds of numbers

Finally, although Sage tries to anticipate what you want, sometimes it does matter how you define a given element in Sage.

- We saw this above with matrices over the rationals versus integers, for instance.
- Here's an example with straight-up numbers.

```
a = 2
```

```
b = 2/1
c = 2.0
d = 2 + 0*I
e = 2.0 + 0.0*I
```

We will not go in great depth about this, either, but it is worth knowing about. Notice that each of these types of numbers has or does not have $I = \sqrt{-1}$, decimal points, or division.

```
print parent(a)
print parent(b)
print parent(c)
print parent(d)
print parent(e)
```

```
Integer Ring
Rational Field
Real Field with 53 bits of precision
Symbolic Ring
Symbolic Ring
```

This is particularly important in the following case. These two things are different types, but are still the same, right?

```
a = 1/5
b = 0.2
```

```
a==b
```

```
True
```

```
b.exact_rational()
```

```
3602879701896397/18014398509481984
```

What the ... ?

This is because Sage "real numbers" are really approximations up to a certain size in machine terms. And the computer only knows binary, so all denominators must be powers of two... even if they are really big ones.

```
2^54
```

```
18014398509481984
```

We can get more precision, of course. But understanding this difference can be crucial when subtle bugs appear because of the difference between rationals or symbolic numbers and "floating-point" numbers.

```
R = RealField(1000)
b = R(b)
print b
```

```
print b.exact_rational()
print b.exact_rational().n(prec=1010)
```

1. Experiment with the list slicing notation, especially the one with two colons. Explain fully what the notation `ls[a:b:c]` means in terms of skips, starts, etc.
2. Do a wide variety of matrix slicing things from documentation. Can you replicate the output of `m[[-1,0],[1,2]]` above with some *method* of `m`? (Hint: use tab-completion.) Discuss the differences between slicing and this method.
3. Write a brief program that takes some square matrix A (of your choice) and checks whether A^2 is the identity matrix.
4. Make this program into both a 'regular' function and a lambda function and check they work.
5. Write a function which, given numerical input, tells whether the square of the input is between 3 and 4.
6. Use filtering to create a list of all primes less than 100 congruent to 1 modulo 4. Use any method you like to create a list of all primes less than 100 which may be written as $p = a^2 + b^2$ for some integers a, b . Discuss.
7. Do Project Euler problem [45](#).
8. Create your favorite graph and your favorite matrix (well, your favorite *nontrivial* ones...) using dictionaries.
9. Completely explain what I have done in the two examples creating graphs using dictionaries and lambdas. (The "join" business is explained [here](#) and [here](#).)
10. Find out about all the things going on in the example about factoring above. Here are a [couple resources](#) about the formatting, but I'll leave you to find out about "if/else" and "try/except". How many new ideas are in it? Where did you look for help? Notice that it's easier to *follow* code than to come up with syntactically correct code - which means you can review other mathematicians' work without having to duplicate it.
11. Do Project Euler [problem 27](#). You will have to use everything here and probably some functions about primality!

12. For a real challenge, do the exercises from [this section](#) of Godsil and Beezer's book on algebraic graph theory.
13. Figure out how to calculate $\sin(1)$ to one hundred bits of precision. How about one thousand? One lakh?
14. Read about the [IEEE floating-point standard](#). Try to create a Sage function which takes a decimal and converts it to this format (mantissa, exponent, and so forth). Then [check your work](#).
15. Make a Sage worksheet or script which shows how to calculate something you care about mathematically (this doesn't have to be something complicated), and which is well enough documented you could teach someone else with it.

--