Programming Tutorial 1 Chennai

Sage Introductory 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 one in a series of tutorials developed for the MAA PREP Workshop "Sage: Using Open-Source Mathematics Software with Undergraduates" (funding provided by NSF DUE 0817071).

Welcome back! First, let's take a few minutes to talk about the homework. Some should be straightforward, and I hope you share your favorite new topics with each other.

However, some should have stretched you (if you have no previous programming experience). In particular, several of them should have required thinking about how to get many, many instances of a Sage command to work together.

In order to do this (and related ideas), we'll need to start stretching our programming muscles. Don't worry, we'll start slowly. We will motivate our examples using matrices and basic situations one might want to handle, but the lessons apply everywhere.

Methods and Dot Notation

First, let's review something that was only introduced last time.

Making a new matrix is not too hard in Sage. We put each row inside brackets, then put the rows themselves (separated by commas) insides brackets.

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

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

(Parenthetical note; we can see how flexible things can be as Sage allows other rings.)

```
print matrix(QQ,[[1,2],[3,4]])
print matrix(RR,[[1,2],[3,4]])
print matrix(GF(3),[[1,2],[3,4]])
```

```
[1 2]
[3 4]
[1.00000000000000 2.00000000000000]
[3.00000000000000 4.00000000000000]
[1 2]
[0 1]
```

Anyway, let's give this matrix of ours a name.

```
A = matrix([[1,2],[3,4]])
A
```

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

So... what can we *do* with a matrix like this? Some commands are available right off the bat. The determinant is one such.

```
det(A)
```

```
-2
```

We call this just a function. But some things are not available this way - for instance a row-reduced echelon form. We can 'tab' after this to make sure.

```
r
```

So, as we've already seen in the previous tutorial, many of the commands in Sage are "methods" of objects. You can think of a method as a function that applies after the fact, sort of like the difference between left and right multiplication when thinking of matrices as group elements or linear transformations.

We access them by typing:

- the name of the mathematical object,
- a dot/period,
- the name of the method, and
- parentheses (possibly with an argument).

This "object-oriented notation" is a huge advantage, once you get familiar with it, because it allows you to do *only* the things that are possible, and *all* such things. Later in the course you will see how to use Python classes to make your own. For now, let's do the determinant again, but this time as a *method*.

```
A.det()
```

```
-2
```

Then we do the row-reduced echelon form.

```
A.rref()
```

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

It is very important to keep in the parentheses.

(Things that would be legal without them would be called 'attributes', but Sage prefers stylistically to hide them, since math is made of functions and not elements of sets. Or so a category-theorist would say.)

```
# Won't work
A.det
```

This is so useful because we can use the 'tab' key, remember!

```
A.
```

Sometimes you will have surprises. Subtle changes in an object can affect what commands are available, or what their outcomes are.

```
A.echelon_form()
```

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

This perhaps surprising outcome is because our original matrix had only integer coefficients, and you can't make the last entry one via elementary operations unless you multiply by a rational number!

In this case, what is going on behind the scenes is that Sage places each matrix in a "matrix space", which is the set of all matrices of a given size with a given ring of coefficients. So we just change the ring.

```
B = A.change_ring(QQ); B.echelon_form()
```

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

As budding programmers, we have to keep track of what is what. In the previous command, we had to rename the changed matrix as "B" because this method returns *a new matrix*; it leaves the old one alone.

```
A.echelon_form()
```

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

This can seem annoying, but as mathematicians we are used to dealing with subtle details. It's just a matter of practice.

Another question is whether one needs an argument. Remember, it's easy to just read the documentation!

Below, let's see whether we need an argument to get a column from a matrix.

```
A.column?
```

It looks like we do. Let's input "1".

```
A.column(1)
```

```
(2, 4)
```

Since what we get is a vector (read the documentation!), we can even *chain* methods to take the dot product

of this vector with itself.

```
A.column(1).dot_product( A.column(1) )
```

```
20
```

See how we have something of the form "A.method1().method2()"? This is the moral equivalent of composition of functions, and it's a popular way to streamline things once you are familiar with what methods you need.

By the way, did you notice that "A.column(1)" gave the SECOND column?

```
A
```

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

Why is that? Shouldn't "1" give the *first* column?

Lists and Loops

Lists

Sage (along with the Python programming language, and many others) begins numbering of anything that is like a sequence at *zero*.

It's very important to remember as it will be useful nearly all the time. To reinforce this, let's formally introduce a fundamental object we've seen once or twice before informally, a Python *list*.

You should think of a list as an ordered set, placed between brackets and separated by commas.

```
ls = [3.1, 4.5, 6.7, -2.8]; ls
```

```
[3.10000000000000, 4.50000000000000, 6.70000000000000,
-2.80000000000000]
```

(Typing two consecutive commands, separated by a semicolon, will do them both.)

Although we won't use this much now, the elements of the ordered set or list can be pretty much anything - including other lists.

```
my_list=[2, 'Ramanujan', [A.column(1), number_of_partitions(9), 5] ];
my_list
```

```
[2, 'Ramanujan', [(2, 4), 30, 5]]
```

You can access any elements of a list easily using square brackets. Just remember that the counting starts at zero.

```
my_list[0]; my_list[1]; my_list[2]
```

```
2
'Ramanujan'
[(2, 4), 30, 5]
```

Can you explain the following two behaviors? They are both quite useful.

```
my_list[-1]
```

```
[(2, 4), 30, 5]
```

```
my_list[3]
```

```
Traceback (click to the left of this block for traceback)
...
IndexError: list index out of range
```

There are lots of advanced things one can do with lists. In the next lecture we will see some more of them.

```
my_list[0:2]
```

```
[2, 'Ramanujan']
```

Loops

However, our main reason for introducing this is more practical, as we'll now see.

- One of the best uses of the computer is to quickly do tedious things.
- One of the most tedious things to do by hand in linear algebra is taking powers of matrices.
- Here we make the first four powers of our matrix 'by hand'.

```
A = matrix([[1,2],[3,4]])
A^0; A^1; A^2; A^3; A^4
```

```
[1 0]
[0 1]
[1 2]
[3 4]
[ 7 10]
[15 22]
[ 37  54]
[ 81 118]
[199 290]
[435 634]
```

This is not terrible, but it's not exactly nice either, particularly if you might want to do something *with* these new matrices.

Instead, we can do what is known as a *loop* construction. See the notation below; it's at least vaguely mathematical.

```
for i in [0,1,2,3,4]:
    A^i
```

```
[1 0]
[0 1]
[1 2]
[3 4]
[ 7 10]
[15 22]
[ 37  54]
[ 81 118]
[199 290]
[435 634]
```

What did we do?

- For (each) i in the list (ordered set) [0,1,2,3,4], (return) A^i.

Yeah, that makes sense. The square brackets created a list, and the powers of the original matrix come in the same order as the list. This is called a *loop*; most computer languages have a construct like this.

The colon in the first line and the indentation in the second line are **extremely** important; they are the basic syntactical structure of Python.

```
for i in [0,1,2,3,4]
    A^i
```

```
Traceback (click to the left of this block for traceback)
...
SyntaxError: invalid syntax
```

```
for i in [0,1,2,3,4]:
A^i
```

```
Traceback (click to the left of this block for traceback)
...
IndentationError: expected an indented block
```

This will undoubtedly come up much more when you learn more advanced material in a couple weeks.

Streamlining

For the curious, it's worth knowing there are quicker ways to make the possible values for i quicker to write. Here are two possible options.

```
for i in [0..4]:
    det(A^i)
```

```
1
-2
4
-8
16
```

```
for i in range(5):
    det(A^i)
```

```
1
-2
4
-8
16
```

Notice that the "range(5)" starts counting at zero and ends *before* reaching five; this is standard behavior. Here is some more.

```
range(3, 23, 2); [3,5..21]
```

```
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

Why not try getting all determinants of all *even* powers of A up to 14 using one of these? (Be careful about where the range ends!)

```

```

There is also a way to get such ranges using Sage types ("srange") but we will not cover it here.

Building Lists

In order to use all those determinants, it would be useful to have a list of *all* of them; printing them out and then using them would require a lot of copy and paste. So let's combine a loop with a simple list method to do it.

First, we need to create an empty list.

```
L = []
```

Now we will combine a loop with one of the few methods that all lists have, "append".

```
for i in range(5):
    L.append( det(A^i) )

print L
```

```
[1, -2, 4, -8, 16]
```

What happened? For each i we wanted, we (consecutively) appended the next determinant to the list. Basic lists do not have many methods, but the ones they do have are useful.

Now one could process the items in L further, or search them for a formula, or whatever you desire.

What happens if I try it again with slightly different syntax?

```
for i in range(5):
    L.append( det(A^i) )
    print L
```

There are *two* strange things! First, the list continues from where we left off. Secondly, we now print the list *each* time instead of at the end. Syntax matters.

List Comprehensions

This all works well. However, after a short time this will seem tedious as well (you may have to trust us on this). It turns out that there is a very powerful way to create such lists in a way that very strongly resembles the so-called set builder notation, called a *list comprehension*.

We start with a relatively easy example:

$$\{n^2 \mid n \in \mathbf{Z}, 3 \leq n \leq 12\}$$

Who hasn't written something like this at some point?

This is a natural for the list comprehension, and can be very powerful when used in Sage.

```
[ n^2 for n in [3..12] ]
```

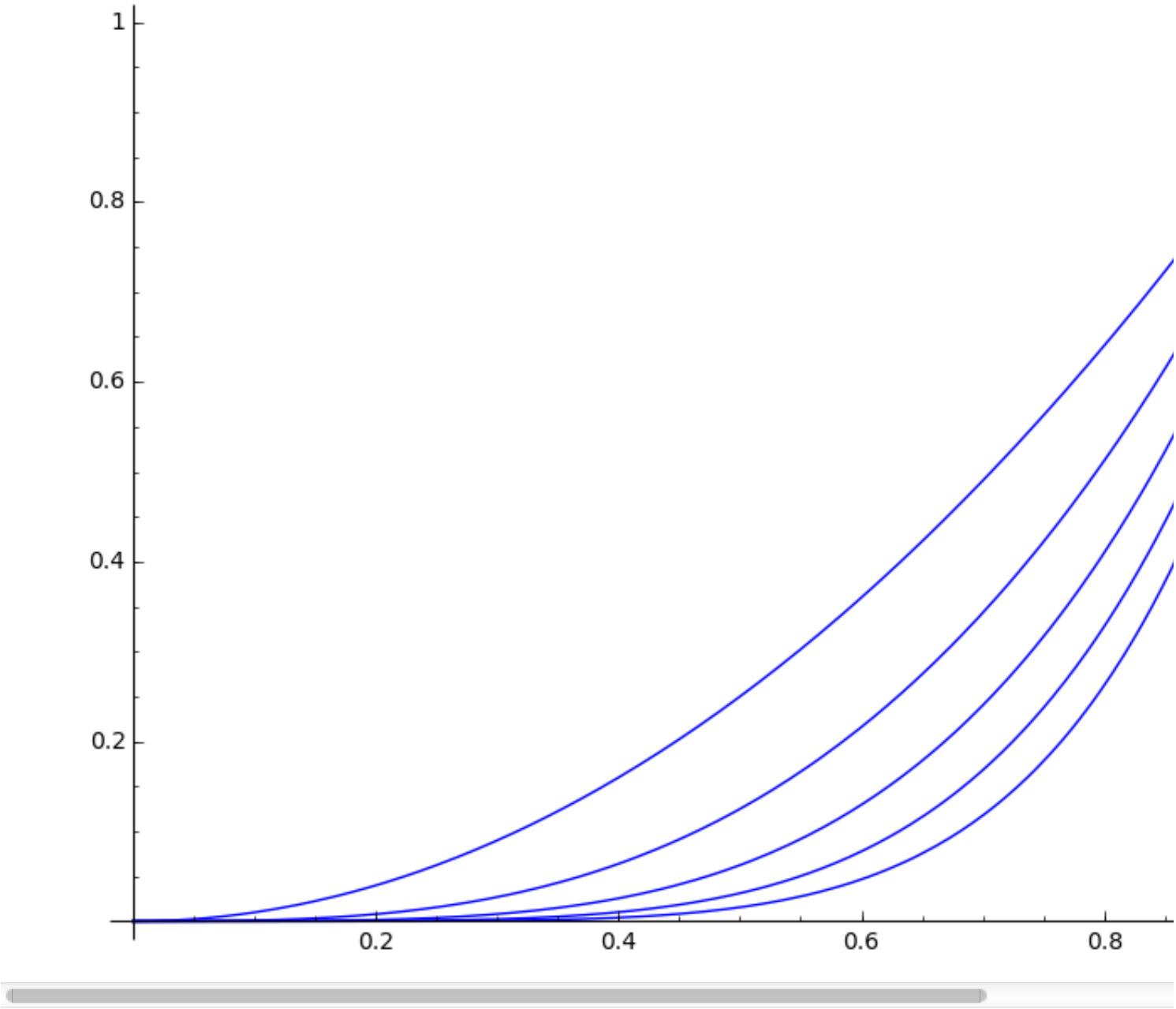
```
[9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
```

That's all there is to it. We sort of turn the construction of the loop around and put it in a list.

- The notation is easiest if you think of it mathematically; "The set of n^2, for (any) n in the range between 3 and 12."

This is phenomenally useful. Here is a nice plotting example.

```
plot([x^n for n in [2..6]],(x,0,1))
```



This sort of construction works for lots of things of mathematical interest, of course.

```
[ det(A^i) for i in [0..4] ]
```

```
[1, -2, 4, -8, 16]
```

```
G = FiniteWeylGroups().example()
[g.reduced_word() for g in G.semigroup_generators() ]
```

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

In this case, we didn't know ahead of time how many elements there would be, but Sage just got the reduced word for all of them!

An application of lists: Tables

Finally, getting away from strictly programming, here is a useful tip that is a nice application of lists.

Suppose you are not just trying to compute determinants, but you want to organize this information for someone else. What can you do to make this easy to visualize?

The answer is the "table" command. First, make a list of the rows you want to display (here, we make each row a list as well).

```
L = [ [i,det(A^i)] for i in [0..4] ]
```

Now just put the list of lists into the table command.

```
table(L)
```

0	1
1	-2
2	4
3	-8
4	16

To make it look nicer, in the notebook we can wrap this in "html":

```
html(table(L))
```

0	1
1	−2
2	4
3	−8
4	16

Even better, we can put a header line on it to make it really clear what we are doing, by adding lists. We've seen keywords like "header=True" when doing numerical precision.

```
html( table( [['i', 'i^2'], [1,1], [2, 4] ] , header_row=True))
```

i	i^2
1	1
2	4

Defining Functions (Extending Sage)

It is often the case that Sage can do something, but doesn't have a simple command for it. For instance, you might want to take a matrix and output the square of that matrix minus the original matrix.

```
A = matrix([[1,2],[3,4]])
A^2 - A
```

```
[ 6  8]
[12 18]
```

How might one do this for other matrices? Of course, you could just always do $A^2 - A$ again and again. But this would be tedious and hard to follow, as with so many things that motivate a little programming. Here is how Python and Sage solve this problem.

```
def square_and_subtract(mymatrix):
    return mymatrix^2 - mymatrix
```

The 'def' command has created a new function called 'square_and_subtract'. It should even be available using tab-completion.

Here are things to note about its construction:

- We use the predefined "def" and then the name we want to give the function.
- The name we want to use for the input is inside the parentheses.
- The indentation and colon are crucial, as with loops.
- We then end with a *return value*, given by 'return'. This is what Sage will give below the input cell.

```
square_and_subtract(A)
```

```
[ 6  8]
[12 18]
```

```
square_and_subtract(matrix([[1.5,0],[0,2]]))
```

```
[0.750000000000000 0.000000000000000]
[0.000000000000000  2.00000000000000]
```

As a technical matter, this is a *Python function*, not a Sage symbolic function (callable expression). That means it does not have access to all the same things; in this case it doesn't matter, but if you wanted to, you could do the following:

```
sq_and_subt(x) = x^2 - x
```

However, often it is better for a function *not* to live in a symbolic algebra world!

What if we are worried about forgetting what this function does? To be fair, we chose a great name for it. Still, just in case, we can provide a documentation string, putting it in triple quotes """.

```
def square_and_subtract(mymatrix):
    """
```

```
    Return `A^2-A`
    """
    return mymatrix^2-mymatrix
```

```
square_and_subtract?
```

Pretty cool! And potentially quite helpful, especially if the function is complicated. The A typesets properly because we put it in backticks `A`.

(For the *real* experts, one can use "raw strings" to include backslashes (say, for LaTeX) in these documentation strings, like r"""\frac{a}{b}""". Look at the documentation for Bessel functions for some great examples.)

A very careful reader *may* have noticed that there is nothing that requires the input 'mymatrix' to be a matrix. Sage will just try to square whatever you give it and subtract the original thing.

```
square_and_subtract(sqrt(5))
```

```
    -sqrt(5) + 5
```

```
square_and_subtract(G.an_element())
```

```
    Traceback (click to the left of this block for traceback)
    ...
    TypeError: unsupported operand type(s) for -:
    'SymmetricGroup_with_category.element_class' and
    'SymmetricGroup_with_category.element_class'
```

This is a typical thing to watch out for; just because you define something doesn't mean it's useful! Sometimes it will be, sometimes not.

Functions are very flexible in what input they can allow or require, as well as what output they give. Below are three examples that show some of this flexibility. There are many other good resources out there.

```
def func1( y, z ):
    return y+z
```

```
def func2( y, z=3):
    return y+z
```

```
def func3( y ):
    return y, y+3
```

Try to define a function which inputs a matrix and returns the determinant of the cube of the matrix. (There

Try to define a function which inputs a matrix and returns the determinant of the case of the matrix. (There are a few ways to do this, of course!)

What's in a Name (Gotchas from names and copies)

Before we finish this tutorial, we want to point out a few programming-related things about names that often trip people up. In both cases, mathematical variables are not the same as computer variables.

The first 'gotcha' is that it's possible to clobber constants! "i" should be the square root of negative one in Sage, but:

```
i
```

```
4
```

Can you figure out why $i = 4$? Look carefully above to see when this happened.

- Or more precisely, when was the *most recent* time it happenend?
- This gives a valuable lesson; *any time* you use a name there is potential for renaming.

This may seem quite bad, but could be quite logical to do - for instance, if you are only dealing with real matrices. It is definitely is something a Sage user needs to know, though.

Luckily, it's possible to restore symbolic constants.

```
restore('i')
i; i^2
```

```
I
-1
```

There is another thing that can happen if you rename things too loosely.

```
A = matrix(QQ,[[1,2],[3,4]])
B = A
```

This actually has just made B and A refer to the same matrix. B isn't like A, it *is* A. The 'copy' command gets around this (though not always).

```
C = copy(A)
A[0,0]=987
show([A,B,C])
```

$$\left[\begin{pmatrix} 987 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 987 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\right]$$

This is very subtle if you've never programmed before. Suffice it to say that it is safest to let each '=' sign

stand for one thing, and to avoid redundant equals.

Homework

Again, try to do this with Sage and its documentation as much as possible. This time, you will have to start programming!

1. If you are on campus, make an account at the Sage notebook server at citron:8080. Remember from the previous tutorial that there is a nice set of instructions for how to get your own copy of worksheets; get a copy of this worksheet and the previous one to try for yourself.
2. In the table above with values of i and i^2, I put those in single quotation marks. It turns out this is what is called a "string". Read some resources about Python strings, like Think Python. How is a string like a list? Find out if it matters if you use double quotes " versus single quotes '.
3. In that same example, what do you think will happen if you use dollar signs around the things inside the strings? Try it!
4. Take a look at some (very) basic tutorials about groups, graphs, or number theory. Pick one type of object, and something you can do to it. Find a way to list all the objects of this sort (pick a finite set!) with this done to it. Example - order of all elements of a finite group.
5. With the same object and thing you can do to it, write a function which does that. Example: Function that, given a group, returns the order of the group. Be sure to add documentation!
6. Find out how to get the following:
    1. The character table of a group
    2. A block diagonal matrix
    3. Both a matrix and a group coming from a graph
7. Do the exercises in the main text about getting determinants of the even powers of the matrix A. What about powers divisible by 3?
8. Define a function which inputs a matrix and returns the determinant of the cube of the matrix. (There are a few ways to do this, of course!)
9. Define a function which inputs a matrix and a power and returns that power of the matrix. What does it do with negative input? Bonus: can you find out how to handle the case when the user (let's say in error) gives a non-integer power?
10. Do Project Euler problem number 1. (You'll probably want to create a list first, then use the "sum" function on it.)
11. Try Project Euler problem number 6. Feel free to use lists and functions if they help you. Hint: try your process with the example given, and other small examples, first!
12. Read about "deepcopy". Think of a mathematical setting where this would be necessary to know about.
13. Try func1, func2, and func3 above and see what they do. Describe how they behave. Then write a function which, depending on input, either takes a number and squares it, or takes two numbers and raises the first one to the second power. (Hint: *default* arguments.)
14. One way to numerically integrate in Sage is like this: "numerical_integral(x^2,0,1)". It returns the answer and an error bound. Find out how to access *just* the answer and *just* the error bound.