Introduction to Sage

INTRODUCTION TO SAGE

AIMS VOLKSWAGEN STIFTUNG WORKSHOP ON COMPUTER ALGEBRA AND APPLICATIONS

DUOALA, CAMEROON, 6-13 OCTOBER 2017

EVANS DOE OCANSEY

RESEARCH INSTITUTE FOR SYMBOLIC COMPUTATION

Saturday, 7th October 2017

The outline of the talk is as follows:

1. What is Sage?

1. An open source system for advanced mathematics.
2. An open source mathematics distribution (like Linux) with *Python* as the glue.
3. A tool for learning and teaching mathematics.
4. A tool for mathematics research.

Mission Statement
Create a viable free open source alternative to Magma, Maple, Mathematica, and Matlab.

1.1. A Brief Overview

- Created in 2005 by William Stein.
- Free and open, GPL license.
- Includes about 100 open source packages.
- Now has around 500,000+ lines of new code, by several hundred mathematician-programmers.

Some of the 100 packages included:

- Groups, Algorithms, Programming (GAP) - group theory
- PARI - rings, finite fields, field extensions
- Singular - commutative algebra
- SciPy/NumPy - scientific computing, numerical linear algebra
- Integer Matrix Library (IML) - integer, rational matrices
- CVXOPT - linear programming, optimization
- NetworkX - graph theory
- Pynac - symbolic manipulation
- Maxima - calculus, differential equations

1.2. Using Other Non-Commercial Softwares

As mentioned earlier, Sage includes about 100 open source packages and these packages can be run separately in the Sage worksheet. There are several ways one can use this. I will only demonstrate a few.

```
s8 = gap('Group( (1,2), (1,2,3,4,5,6,7,8) )')
```

```
%gap
S := Group( (1,2), (1,2,3,4,5,6,7,8) )
```

```
maxima('lsum(x^i, i, [1, 2, 7])')
```

```
%maxima
lsum (x^i, i, [1, 2, 7]);
```

```
%python
map(lambda x : x**2, [1,3,5])     # note the operation ** and not the same as
^ in Python but in Sage they are.
```

```
%singular
ring R = 0,(x,y,z), lp; R;
```

2. Help Inside Sage

There are various ways to get help for doing things in Sage. Here are several common ways to get help as you are working in a Sage worksheet.

2.1. Documentation

Sage includes extensive documentation covering thousands of functions, with many examples, tutorials, and other helps.

- One way to access these is to click the "Help" link at the top right of any worksheet. This page has lots of useful commands for the notebook. At the top it has a list of documents; you can click your preferred option at the top of the help page.
- They are also available any time online at the Sage website, which has many other links, like video introductions.
- The Quick Reference cards are another useful tool once you get more familiar with Sage.

Our main focus in this tutorial, though, is help you can immediately access from within a worksheet, where you don't have to do *any* of those things.

2.2. Tab completion

The most useful help available in the notebook is "tab completion". The idea is that even if you aren't one hundred percent sure of the name of a command, the first few letters should still be enough to help find it. Here's an example.

- Suppose you want to do a specific type of plot - maybe a slope field plot - but aren't quite sure what will do it.
- Still, it seems reasonable that the command might start with `pl`.
- Then one can type `pl` in an input cell, and then press the tab key to see all the commands that start with the letters `pl`.

Try tabbing after the `pl` in the following cell to see all the commands that start with the letters `pl`. You should see that `plot_slope_field` is one of them.

```

```

3. Complete Access to Source Code

Unlike other commercial softwares, SageMath gives its users a complete access to the source code. For example let's see the source code for the Sage function `is_square` and `is_squarefree`. Both are in the same file. All we need to do is to type the name of of the function and after a double question mark and press Enter or hit the tab key.

```
is_square??
```

Try 1:

Find the source code for the Sage functions derivate and factor.

```

```

5. Object Oriented Paradigm

The object-oriented programming paradigm consists in modelling each physical or abstract entity one wishes to manipulate by a programming language construction called an *object*. In most cases, as in Python, each object is an instance of a *class*. For example, the rational number $\frac{17}{35}$ is represented by an object which is an instance of the **`Rational`** class:

```
m = 17/35
```

```
type(m)
```

```
<type 'sage.rings.rational.Rational'>
```

Note that this class is really associated to the object $\frac{17}{35}$, and NOT to the *variable* m in which it is stored:

```
type(17/35)
```

```
<type 'sage.rings.rational.Rational'>
```

More precisely, an object is a part of the computer memory which stores the required information to represent the corresponding entity. Thus a class in turn defines two things:

1. the *data structure* of an object, i.e., how the information is organised in memory. For example, the Rational class specifies that a rational number like $\frac{17}{37}$ is represented by two integers: its numerator and its denominator
2. its *behaviour*, in particular the available *operations* on this object: how to obtain the numerator of a rational number, how to compute its absolute value, how to multiply or add two rational numbers. Each of these operations is implemented by a method (here respectively `numer`, `abs`, `__mult__`, `__add__`)

In Sage, when we instantiate an object and assign it a variable, we can have access to all methods or operations which are available for object. All we need to do is to enter the object, then enter the period sign and then press the tab key. For example let's see the methods available for the object $\frac{17}{35}$ assigned to the variable m.

```
m.
```

$$5^{-1} \cdot 7^{-1} \cdot 17$$

Try 2: Create a 4×4 matrix space over the polynomial ring $\mathbb{F}_5[x]$ and pick one of the check the methods that are available to an element in this space.

```
%hide
Z5 = GF(5); show(Z5)
F.<x> = Z5[]; show(F)
show(PolynomialRing(Z5, 'x'))
M = MatrixSpace(F, 4, 4); show(M)
show(M.random_element())
p = M.random_element()
```

$$\mathbf{F}_5$$

$$\mathbf{F}_5[x]$$

$$\mathbf{F}_5[x]$$

$$\mathrm{Mat}_{4\times 4}(\mathbf{F}_5[x])$$

$$\begin{pmatrix} 4x^2+2x+2 & 3x^2+1 & x^2+4x+3 & 3x^2+3x+2 \\ 2x^2+3x+1 & x^2+2x+4 & x^2+4x & x^2+x+3 \\ 2x^2+3x+2 & 2x+4 & 2x^2+x+4 & x^2+4 \\ 2x^2+2x+1 & 4x^2+2x+4 & 4x^2+4x+1 & x^2+3 \end{pmatrix}$$

Try the same problem for the number field $\mathbb{Q}(\sqrt{-5})$.

6. Expressions and Computational Domains

Some computer algebra system requires the user to always specify the domain in which he or she wants to work. Sage in some sense does this for you. It has a ring called Symbolic Ring and one can compute in this domain. It is denoted by `SR` any it's elements are symbolic expressions. Any expression that we define is in this ring unless otherwise, we specify the domain. The symbolic expression x is already defined in Sage for us. There are other computational domains as well like integers, rationals, floating points polynomial rings, etc. Let us consider the factorisation of the following polynomial expression:

```
x = SR.var('x')                          # Create a symbolic expressions x and
assigns it to the variable x.
p = 54*x^4+36*x^3-102*x^2-72*x-12        # Sage considers this as a symbolic
expression which we humans sees it as a polynomial.
show(factor(p))                          # Factor p over the SR domain.
```

$$6\left(x^2-2\right)(3\,x+1)^2$$

Sage cannot know if we wish to factor p as a product of polynomials with integer coefficients, or with rational coefficients. In order for Sage to do this, we will have to specify which mathematical set p lives. Say for instance $p \in \mathbb{Z}[x]$. Then

```
R = PolynomialRing(ZZ,'x'); R
```

```
Univariate Polynomial Ring in x over Integer Ring
```

```
q = R(p); q
```

```
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
```

```
parent(p), parent(q)
```

```
(Symbolic Ring, Univariate Polynomial Ring in x over Integer Ring)
```

As a consequence, its factorisation is uniquely defined:

```
show(q.factor())
```

$$2 \cdot 3 \cdot (3x + 1)^2 \cdot (x^2 - 2)$$

Try:

Let us proceed similarly in the rational field, polynomial ring over the finite field $\mathbb{F}_5$, and polynomial ring over the Number field $\mathbb{Q}(\sqrt{2})$.

7. Loops

Sometimes we might want to do an operation repeatedly for a number of times. This can be done with the keyword for or while. Let's see each of them separately.

7.1. For Loops

We use for loops when the number of iterations is already known. The construction is as follows:

```
for <loop variable name> in <iterable>:
    <indented block of code>
```

For example. Let us compute determinant of the power of some matrix defined over our own domian of choice.

```
A = matrix(QQ, [[1,2/3], [2/5, 7/9]]); A
```

```
[  1 2/3]
[2/5 7/9]
```

```
for i in [0..3]:
    print det(A^i)
```

```
1
23/45
529/2025
12167/91125
```

Try:

Consider the sequence (u_n) defined by

$$n_0 = 1, \quad \forall\, n \in \mathbb{N},\ u_{n+1} = \frac{1}{1 + u_n^2}.$$

Write a for-loop to approximate u_n for $n = 20$.

```
%hide
U = 1.0 # or U = 1. or U = 1.000
for n in [1..20]:
```

```
    U = 1 / (1 + U^2)
U
```

```
0.682360434761105
```

7.2. While Loops

The while loop, as its name says, executes instructions while a given condition is fulfilled. They have the following constructs:

```
while <condition>:
    <indented block of code>
```

Here is a `while` loop version of the `for` loop just above

```
i=0
while i <= 3:
    print det(A^i)
    i = i+1
```

```
1
23/45
529/2025
12167/91125
```

Try: Let us compute the sum of the squares of non-negative integers whose exponential is less or equal to 10^6. That is

$$\sum_{k\in\mathbb{N} \text{ and } e^k\leq 10^6} k^2$$

```
#%hide
ans, k = 0, 0
while e^k <= 1e6:
    ans = ans + k^2
    k += 1
ans
```

```
819
```

```

```

8. Conditionals

The decision structure is to run something in the event that a condition is true. Decision structures evaluate multiple expressions which produce TRUE or FALSE as result. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

That said, these comparisons can be placed inside of an **if** statement. Such statements have the following form:

```
if <condition>:
    <indented block of code>
```

The indented code will only be execute if the condition evaulates to *True*, which is a special boolean value.

```
num = 4
if num % 2 == 0 and num != 0:
    print num/2
```

```
2
```

The **if** statement can be combined to great effect with a corresponding **else** clause.

```
if <condition>:
    <if-block>
else:
    <else-block>
```

When the condition is *True* the if-block is executed. When the condition is *False* the else-block is executed instead.

```
num = 5
if num % 2 == 0 and num != 0:
    print num/2
else:
    print 3*num + 1
```

```
16
```

Using the keyword **elif** we can even add more conditions.

```
num = 0
if num % 2 == 0 and num != 0:
    print num/2
elif num % 2 == 1 and num != 0:
    print 3*num + 1
else:
    print('I am zero.')
```

```
I am zero.
```

Functions and Procedures

One can use Sage to define procedures or functions using the keyword def whose syntax will be detailed below for both procedures and functions. We call a function (resp. procedure) a sub-program with zero argument, one or several arguments which returns (resp. does not return) a result. The construct for a Sage/Python procedure is as follows:

```
def <procedure name>():
    <procedure body>
```

For example, let's write a Sage/Python procedure that prints 5

```
def i_am_a_procedure():
    """
    This procedure only prints `5`.
    """
    print 5
```

On the other hand, that of a Sage/Python function is as follows.

```
def <function name>(<argument>):
    <function body>
    return <local variable>
```

Let's create a Sage/Python function that returns 3.

```
def i_am_a_function():
    """
    This function returns `3`.
    """
    return 3
```

Try:

Write a function that takes two arguments a and b and returns their sum of squares.

```

```

```
    True
```

```
%hide
def sum_of_squares(a=3, b=4):
    """
    Assumes that `a` and `b` are integers and returns `a^2 + b^2`.
    """
    assert str(parent(a)) == 'Integer Ring' and str(parent(b)) == 'Integer
Ring'

    return a^2+b^2
```

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.

```
sum_of_squares_lambda = lambda a=3, b=4 : a^2 + b^2
```

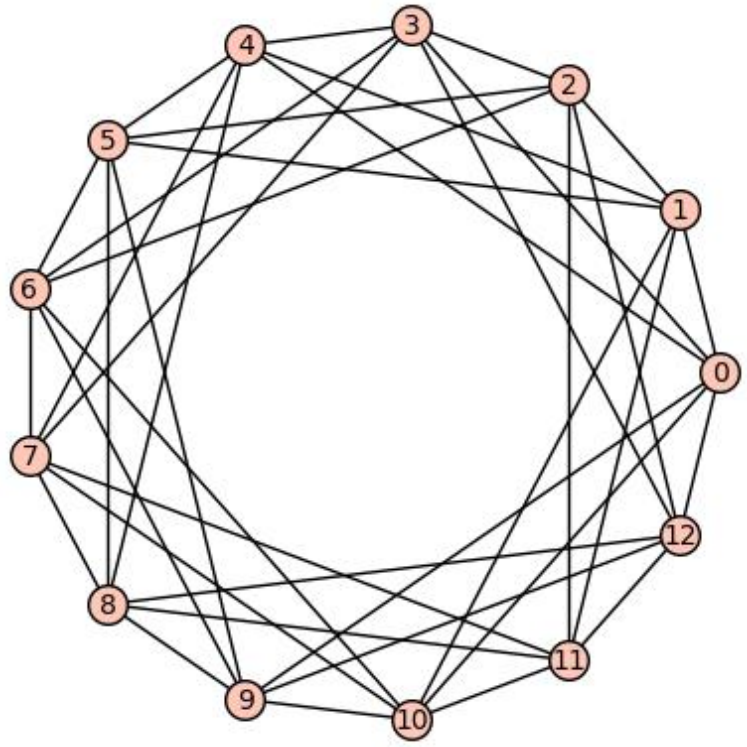
```
sum_of_squares_lambda()
```

```
    25
```

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

Here is a cool example of using this. With just one line of code, we construct and show a Paley graph. You don't need to know what this is to agree it is powerful.

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



Try:

Write a lambda function that takes two arguments `a` and `b` and returns their sum of squares.

```
```

Recursive Functions

One of the greatest features of functions is that they may call themselves from withing their own bodies! This is known as recurssion.

Try:

Write a recursive function to compute $1 + 2 + 3 + \cdots + n$ for $n \geq 1$.

```
%hide
def add_first_positive_n(n):
    """
    Assumes that `n` is a positive integer and returns the sum of the first
`n` integers.
    These numbers are the so called triangular numbers.
    """
    assert n >= 0 and str(parent(n)) == 'Integer Ring', "%s is not a non-zero
integer" %n
    if n == 0 or n == 1:
        return n
    else:
        return n + add_first_positive_n(n-1)
```

```
add_first_positive_n(10)
```

```
55
```

Try:

Implement the Collatz Problem

$$c(n) := \begin{cases} \frac{n}{2} & \text{if } n \text{ is even} \\ 3\,n+1 & \text{if } n \text{ is odd and } n > 1 \\ 1 & \text{if } n = 1. \end{cases}$$

using recursion for $n \in \mathbb{N} \setminus \{0\}$.

```
%hide
def collatz_recursion(num):
    """
    Assumes that `num` is a positive integer and confirms collatz conjecture.
That is, for any
    positive integer it will return 1 at some point.
    """
    assert num > 0 and str(parent(num)) == 'Integer Ring', "%s is not a
positive integer."%num

    if num == 1:
        return 1
    elif num > 0 and mod(num, 2) == 0:
        return collatz_recursion(num/2)
    elif num > 1 and mod(num, 2) == 1:
        return collatz_recursion(3 * num + 1)
```

Try:

Write a Sage/Python procedure using while loop that implements the collataz problem but this time around, print the sequences that you get. So for example if the name of my function is `collatz_procedure` and I call `collatz_procedure(5)` it print $5, 16, 8, 4, 2, 1$.

```
%hide
def collatz_procedure(n):
    print(n)
    while n > 1:
        if n % 2 == 0:
            n = n/2
            print(n)
        else:
            n = 3 * n + 1
            print(n)

collatz_procedure(5)
```

```
5
16
8
4
2
1
```

```
```

Symbolic Expressions

Symbolic expressions in the symbolic ring can also converted into callable functions.

```
(r, t) = var('r, t')
FV = 100*e^(r*t)
```

```
f = FV.function(r, t)
```

We can also functions defined as this.

```
y = function('y')(x)     # This sort of function definition will be needed by
one of the guest speakers Georg.
```

```
show(y + diff(y))
```

$$y(x) + \frac{\partial}{\partial x} y(x)$$

Data Structures

Lists

The computer language for Sage is essentially that of Python, but with some sweetening to make it more mathematical.

- So we can make lists!
- And lists start numbering at ZERO.

Remember, a list is basically an ordered set, placed between brackets and separated by commas.

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

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

The elements of the ordered set or list can be pretty much anything - including other lists.

```
my_list=[2, 'Ramanujan', [1,2,3] ]; my_list
```

```
[2, 'Ramanujan', [1, 2, 3]]
```

```
null_list = []; null_list
```

```
[]
```

Recall how to access elements of the list.

```
my_list[1] # Get the second element in the list my_list. Sage/Python indexing
starts from zero
```

```
my_list[-1]    # Get the last element in the list my_list
```

```
my_list[3]    # There is no 4th element in the list thus the error.
```

Do you remember how to access the methods available to any object defined in Sage? Let us get the methods available for list objects

```
my_list.
```

Try:

Try to turn the procedure you wrote for printing the collatz sequence into a function that returns a list where this list now contains the collatz sequence.

Tuples

```
my_tuple = (1,2,3)
```

```
null_tuple = ()
```

```
my_tuple[0]
```

```
1
```

Dictionaries

A Python dictionary is a unordered collection of key-value pairs. Dictionaries are likely the most useful data type in Python you will use in everyday programming. The key is a way to name the data, and the value is the data itself. Here's a way to create a dictionary that contains all the data in our data.dat file in a more sensible way than a list.

```
N = {'a': ['b', 'f'], 'b': ['a', 'c', 'f'], 'c': ['b', 'd'], 'd': ['c', 'e',
'f'],
        'e': ['d', 'f'], 'f': ['a', 'b', 'd', 'e', 'g'], 'g': ['f'] }

def random_walk(steps):
    """
    Input: steps -> integer

    Output: (nodes_visited, count_nodes_visited, nodes_visited_most) -> tuple

    This function takes an integer as input and returns a tuple, (x,y,z) with
    entries defined as follows:

    x -> list of nodes visited
    y -> dictionary with nodes as keys and values as number of times node has
         been visited
    z -> list of two element (u, v) tuples where:
         u -> node visited frequently and
```

```
        v -> number of times node is visited.
    """

    try:
        # Initial position choosen randomly
        position = choice(N.keys())

        # Histroy of nodes visited in order
        nodes_visited = [position]

        # Initialise frequency of each node to zero.
        count_nodes_visited = dict( zip(N.keys(), [0] * len(N.keys())) )

        for step in xrange(steps):
            # Get node's new position of and append to nodes_visited
            position = choice(N[position])
            nodes_visited.append(position)

            # Check if position has already been visited
            if position in count_nodes_visited.keys():
                # Increment frequency of position
                count_nodes_visited[position] += 1

            # Set self.position to be current position
            position = position

        # Get maximum node visited
        max_freq = max(count_nodes_visited.values())

        # Get nodes mostly visited
        nodes_visited_most = [(key, count_nodes_visited[key]) \
                                for key in count_nodes_visited.keys() \
                                if count_nodes_visited[key] == max_freq]

    # Raise TypeError if input argument steps is not an integer
    except TypeError:
        print "TypeError: second argument must be an integer"

    # Do nothing this if no error occurs.
    else:
        pass

    # This is returned in case.
    finally:
        return nodes_visited, count_nodes_visited, nodes_visited_most
```

```
random_walk(20)
```

```
(['c',
  'b',
  'f',
  'a',
  'b',
  'c',
  'd',
```

```
        'e',
        'f',
        'g',
        'f',
        'd',
        'f',
        'g',
        'f',
        'e',
        'f',
        'a',
        'b',
        'f',
        'g'],
       {'a': 2, 'b': 3, 'c': 1, 'd': 2, 'e': 2, 'f': 7, 'g': 3},
       [('f', 7)])
```

```
%hide
def collatz_sequence(n):
    l = [n]
    while n > 1:
        if n % 2 == 0:
            n = n/2
            l.append(n)
        else:
            n = 3 * n + 1
            l.append(n)
    return l, len(l)-1
```