Sage tutorial

Prof. Shahed Sharif

1 Getting started

Open up a terminal and type `jupyter notebook`. If that doesn't work, you can try `sage -n jupyter`. Windows users may have to consult `https://doc.sagemath.org/html/en/installation/launching.html` to troubleshoot.

In the upper right is a drop-down labelled "New". Select "SageMath". This will create a new Sage notebook file "Untitled.ipynb" in the directory you ran Sage. Click on "Untitled" at the top to change the name to whatever you'd like.

As you're reading below, you'll see stuff that looks like programming. When you do, type out each line *exactly as you see it* in an empty cell. To execute, press `Shift + Enter`. (Or use `Ctrl + Enter` if you don't want to create an additional cell.) If you want to enter additional lines within a shell, just hit `Enter`. If you get an error, it is overwhelmingly likely that you typed something incorrectly—the error message should tell you what went wrong. Pay particular attention to tabs at the beginnings of lines—these are crucial to how Python functions! The interface will try to anticipate the correct indentation, but it will not always be correct. Backspace will *dedent*, i.e. decrease the indentation level by 1.

2 Basics of programming

What a computer program does—and by extension, how we understand computer programming—can be divided up into five parts:

I/O. This stands for "input/output". That is, the program will often ask for some information from the user (input), and at some point will display results (output). This can be very complicated if, for example, your program is a video game. For us, I/O will be very simple.

Data. A program stores and manipulates data. Data is, broadly, stored in *variables*, of which there are a number of different types. A program can change the content of a variable, and also read the contents of a variable.

Arithmetic. Programs can do arithmetic. This at a minimum means the 4 standard arithmetic operations, but Sage can natively do a lot more.

Booleans. A program can determine the truth or falsity of certain types of statements. A *Boolean value* is essentially either "true" or "false", and a *Boolean function* is something that inputs a bunch of statements and, based on the truth of those statements, outputs a Boolean value.

Control flow. Control flow refers to the order in which the computer executes commands. One way of looking at this is that control flow instructs the computer what it should do next. The most important part of control flow is *iteration*; that is, doing the same or similar tasks or calculations repeatedly. Computers are much better suited to iteration than humans!

2.1 Putting the parts together

Any algorithm can be put together out of these constituent parts. When writing a program, your first task should be to figure out how to write your algorithm out of these parts. **You should do this away from the computer!**

Once you've done that, let's look at how the various parts of a computer language are implemented in Sage. (Note that Sage is built on Python, so much of the syntax is inherited from Python.) Remember that after entering each command below, press `Shift + Enter` to execute.

3 Arithmetic

```
  5+3
  5-3
  5*3
  5*(2+3)
  5/3
  5//3
```

Note the difference between integer division and regular division!

There are two more useful operations; see if you can determine what they do:

```
  2^3
  17%5
```

To work in other rings, you have to define the ring first. The # denotes a comment; you do not have to type that in!

```
R.<t> = PolynomialRing(QQ)
# R = Q[t], where Q denotes rational numbers
f = 3*t^2 - t + 4
g = t + 1
f*g
f%g
```

4 Data

There are many data types in Sage: integers, integers mod 5, complex numbers, matrices, polynomials, etc.

```
x=5
x
print(x)
print('x')
x+3
(x,y) = (3,5)
y
```

Note that = denotes *assignment*; that is, the expression on the right is evaluated, then assigned to the variable on the left.

```
x=3
x=x+1
x
x+=1
x
x-=2
x
g *= t
f *= t
gcd(f,g)
```

Lists are finite sequences of other data types. Elements of a list can be numbers, strings, or even other lists.

```
li=[1,2,'apple']
li[0]
li[2]
len(li)
li[len(li)]
li[len(li)-1]
li[1:2]
li[1:3]
li[1:]
li[:2]
m=li+li
m
```

As you can see, list operations are very intuitive.

We can define ideals using lists of generators.

```
I = R.ideal([f, g])
I
I.gens()
```

5 Booleans

We start with a single variable assignment ($x = 3$) and proceed with some truth tests and more complex boolean functions (such as AND and OR). Note the difference between = and ==.

```
x=3
x==3
x==2
x<3
x<4
x!=4
x!=3
x==3 and x<4
x==3 and x<2
x==3 or x<2
not (x==3 or x<2)
not x!=4
```

One handy trick that comes up when using control structures is that `0` and `[]` are equivalent to `False`, while all other values are considered `True`.

6 Control flow

Here's a simple loop:

```
for i in range(10):
    print(i)
```

I used a single `Tab` to indent the second line.

Observe the colon, indentation, and how the output relates to the bounds in the **`range`** declaration. The notebook should take care of the indentation for you, but only to a certain extent: to end the indentation, you have to press backspace on a new line.

Another way of making loops is with **`while`**.

```
i=0
while i<10:
    print(i)
    i+=1
```

(Remember to hit enter twice after the last line!) What happens if you do the same program, but with the last two lines reversed?

A common construction is to combine creating, reading, or changing a list with a loop. In the case of a **`for`** loop, we might have something like `range(0,len(listname))`.

The last control structure is the **`if/else`** declaration.

```
if i==11:
    print('bye')
    print('time to go')
else:
    print('twelve')
    print('oops too late')
```

The **`else`** is optional. One can also mimic the two loops above by combining **`if`** with recursion, but in Python it doesn't make sense to do so.

Exercise 1. Compute the 100th triangular number without the formula. Do it in two different ways: using a **`for`** loop and a **`while`** loop.

Exercise 2. Given that the third Fibonacci number is 2, compute the 30th Fibonacci number. (Hint: you will need more than one variable.)

Exercise 3. Create a list of the numbers 1 through 100; by "list", I mean the data type, so `[1, 2, ..., 100]`. Start with `li = []`.

7 Functions and syntax

We now figure out how to put everything together to write a program. As an example, we'll recreate multiplication of positive integers; that is, given positive integers `a` and `b`, we want to write a program which outputs the product of the two using only addition.

7.1 Writing down the algorithm

The key to writing a program is answering the questions

- "What are we looping?" and
- "How do we know when to stop looping?"

We should first do this on paper! We write down what we want in plain English:

1. Given $a, b \in \mathbb{N}$.
2. Write down a on a paper b times.
3. Add together.

Okay, not too bad. However, this doesn't answer either of our questions. For the first question, in step 2 we are looping adding a each time. For the second question, we stop once we've done b iterations. To implement the loop, we can use **`for`** or **`while`**; for this example, I will use **`while`**.

1. Given $a, b \in \mathbb{N}$.

2. Start a running total at 0.
   (a) Add a to our running total.
   (b) If we've done this b times, stop the loop.
   (c) If not, go back to (a).
3. Output the running total.

To make the loop work, we've introduced two new quantities: the running total, and the number of times we've been in the loop. The first quantity we can call `p` for product, while the second quantity is a counter related to `b`. The counter can start at 0 and work its way up to `b`, or can start at `b` and count down to 0. I will choose the latter, for reasons that will become apparent. When we do this, the termination condition is that the counter reaches 0.

7.2 Typing up the program

To type up the program, press an `Enter` at the end of each line to get to the next line, up until you are done with the program. Once you are, press `Shift + Enter` to execute the program definition. Note that all flow declarations end in a colon, and all instructions inside it are indented. The same formalism holds for functions, as you can see below.

```
def dumb_multiply(a,b):
    """Multiply positive integers a and b."""
    p = 0
    while b!=0:
        (p,b) = (p+a, b-1)
    return p
```

Note that our counter is `b` itself! To understand how the program works, I recommend choosing small values of input and try executing the program yourself, on a piece of paper (not with the computer).

Once you've hit `Shift + Enter`, you've defined the function `dumb_multiply` as a function of two arguments. On the next line of the notebook you can now type, for example, `dumb_multiply(5,8)`, and hit `Shift + Enter` to execute.

Note that the program ends with a **`return`** statement. Run on its own, **`return`** acts just like **`print`**. However, **`return`** is superior because the output of **`return`** can be used as the input of another program. Indeed, complex programs are typically split into pieces, where one function will call the results of another. Note however that a **`return`** statement, unlike **`print`**, ends the enclosing function immediately.

Lastly, the variables `a` and `b` are *localized*, so even though their values are changed inside the function, they are not changed outside of the function. That is, if we did (with `Shift + Enter` after each line)

```
a=18
b=15
dumb_multiply(18,15)
b
```

then the value of `b` is still 15, even though the value of `b` appears to change inside `dumb_multiply`.

7.3 Other commands

There's a function-like construction in Python (and many other languages) called a *method*. A method is a function associated to some data. You've seen this before with ideals: an ideal has a `gens` method, which lists the defining generators. The syntax is always `dataobject.method()`. (Methods can sometimes take inputs, which would be put inside the parentheses; most of the time we will just leave the parentheses empty.) Here are some more useful methods:

```
f.degree()
f.leading_coefficient()
```

Plotting is pretty intuitive in Sage. Here's an example; note that since `y` did not have any meaning until now, we have to define it.

```
u = var('u')
implicit_plot(u^2 - t^3 + t, (t, -2, 5), (u, -5, 5))
p = parametric_plot((2*cos(t), 3*sin(t)), (t, 0, 2*pi))
show(p)
```

Sage can also do 3d plots. Look online for documentation and examples.

8 Comments and doc strings

Look at the second line of the `dumb_multiply` function above. The text inside the triple quotes is called the *doc string* for the function. It provides a brief description of the function, and is always a good idea to include. If you hover over the name of a function later, a tooltip should appear with the first line of the doc string.

Any text beginning with the hash symbol `#` is considered a *comment*; that is, it is ignored by Sage. This is good to explain anything in your program that might be hard to understand otherwise.

Exercise 4. Write a program that inputs nonzero $f \in \mathbb{Q}[x]$ and outputs the leading term.

Exercise 5. Write a program that inputs $f \in \mathbb{Q}[x]$, $n \in \mathbb{N}$ and lists all integer roots r of f in the range $[-n, n]$.

Exercise 6. Write a program `division_algorithm` that implements the division algorithm in $\mathbb{Q}[x]$. Do not use `%` or `//`; use only `+,-,*,/, degree`, and `leading_coefficient`. For instance, you should have

```
division_algorithm(x^3+1, x+2) = (x^2 - 2*x + 4, -7).
```

(Note that Sage will treat the quotient of two monomials as a rational function even if it's a polynomial. The easiest fix *coerce* it into the polynomial ring: if that ring is `R`, `R(q)` treats it as an element of `R`.)

Exercise 7. The algorithm for $\gcd(f, g)$ is the same as over $\mathbb{Z}$; namely, define a sequence r_i with $r_0 = f$, $r_1 = g$, then recursively define r_{i+1} as the remainder when we divide r_{i-1} by r_i. Use `division_algorithm` to write `mygcd` (since `gcd` is taken). Your output should agree with the built-in `gcd`; demonstrate.

9 Rules for turning in programs

When turning in programs,

- do not use any forbidden commands;
- upload your notebook file (which should have a `.ipynb` extension) to Gradescope;
- make sure code runs without errors, returns the correct results, and passes any assigned tests;
- make sure all programs have doc strings;
- copy and paste previously written code where appropriate; and
- when required, include a proof that the code is correct with your written homework.

When you use previously written programs, clearly state what those programs do in the doc string; specifically, the input and output of the programs. Of course, that means that you should save your previous code, as it will be useful throughout the course.

For homework, turn in exercises 4–7.

10 Further reading

This document is only meant to get you started. I encourage you to look online for more details on programming in Python/Sage! I'd especially suggest looking into lists (particularly *list comprehension*), printing, and the *dictionary* data type.