Contents

1. Writing a function into Sage 1
2. Committing and Forgetting the Changes 3
3. Sage Trac 3
4. The Fluff 5
5. Installing and Initializing git 7
6. Installing Sage using git 7
7. The Branches of git 7

This is intended to be a beginner's guide to developing Sage. There is plenty more information about git, git trac, and, of course, Sage online.

If you have not installed git on your computer jump ahead to section 5.

1. Writing a function into Sage

(1) Open your terminal and navigate to your Sage directory. Depending on your method of download this will probably look like one of the following:

(a) If you installed Sage via git

```
[user@localhost:∼]$ cd sage
```

(b) If you installed Sage via the Sage website

```
[user@localhost:∼]$ cd sage-*.*
```

where the *'s are replaced with the correct version number.

(c) If you are using Sage on the k8s server:

```
$ cd /Src/sage-BLAH
```

where BLAH is replaced by your username, or whatever you chose to name your file as.

(2) Use git to checkout a branch called `test`:

```
[user@localhost:∼/sage] git checkout -b test
```

There is a bit going on here if you don't know how git works. In that case, go ahead and check out Section 7.

(3) Open up a new file using your favorite text editor.

(4) Save the file as `test.py` in the

```
/sage/src/sage/rings/number_field
```

or the

```
/sage-*.*/src/sage/rings/number_field
```

directory, again depending on install method.

Important: This is not necessarily the best form. Generally you want to place your file in the most relevant directory to it.

(5) Write some sort of function in this file in the python language. For the sake of it, let's say write a function called `Fibonacci` that inputs an integer n and returns the n-th Fibonacci number. Another option would be to write a function called `Plus` which inputs two numbers and returns their sum.

(6) Now return to your terminal and open Sage by typing

```
[user@localhost:~/sage]$ ./sage
```

Try using your function.

(7) Oh that was rude, your function is nowhere to be found.

Go ahead and quit Sage.

```
sage:  quit
```

(8) Ok, now let's rebuild Sage so that maybe it can recognize your file. If you're on the k8s server use

```
$ make build
```

If you are on your personal computer you could use

```
[user@localhost:~/sage]$ ./sage -b > output 2>&1
```

the last part `2>&1` says, forward `stderr` to `stdout`, and the first part `> output` says forward `stdout` to `output`. This will rebuild Sage, and may take a bit of time, but it will be nothing compared to the original `make` process.

(9) Let's try again, open Sage and use your function.

Wait, still rude of me, it's still not there.

But everything is ok this time. Import it using the typical Python:

On your own machine:
```
sage:  from src.sage.rings.number_field.test import Fibonacci
```

On the k8s server:
```
sage:  from sage.rings.number_field.test import Fibonacci
```

Of course import the function you created.
Now you can use your function within Sage.

(10) Well that was a lot of work. There are plenty of Sage functions that don't require importing. Let's do that. Quit Sage again. Now open the file `/src/sage/rings/number_field/all.py` in you chosen text editor.
At the very bottom of the file add the line

```
from .test import Fibonacci
```

This will tell Sage to import your function.

(11) Go ahead and do the `make build` or `./sage -b`, open Sage and use your function straight away without importing it.

(12) Your turn! Write another function into your file `test.py` and include it into Sage in both possible ways.
Try writing a function that makes use of another Sage function or object.

2. Committing and Forgetting the Changes

(1) Let's tell git what files we have been editing on the branch `test`.

```
[user@localhost:~/sage]$ git add src/sage/rings/number_field/test.py
[user@localhost:~/sage]$ git add src/sage/rings/number_field/all.py
```

(2) We then have to tell git that it should take a look at the files we added and remember the changes.

```
[user@localhost:~/sage]$ git commit -m "Added a function called Fibonacci"
```

The `-m` option is required and indicates that the following string describes the changes that were made in this commit. Be as descriptive but precise as possible in these messages. You want to be able to look back and remember what you have done.

(3) Ok so now the test branch is going to remember our function we added. So let's return to the `develop` (or `master`) branch that we started on

```
[user@localhost:~/sage]$ git checkout develop
```

(4) Again build Sage, open it and try to run Fibonacci.
It should be gone now. What the what, thanks git.
Anyway, let's keep this test branch around for a little while so we can use it later.

3. Sage Trac

(1) Now that we're comfortable with developing Sage, let's take it to the next level by connection to trac. If you do not yet have git-trac on your machine, direct yourself to your ~ directory and clone the git repository:

```
[user@localhost:~/sage]$ cd ~
[user@localhost:~]$ git clone https://github.com/sagemath/git-trac-command.git
[user@localhost:~]$ source git-trac-command/enable.sh
```

(2) Return to your Sage directory so we can configure your Sage Trac account.

```
[user@localhost:~]$ cd sage
[user@localhost:~/sage]$ git trac config --user USERNAME --pass 'PASSWORD'
```

except use your own username and password here.

Note that you can use git trac to look at tickets in a read only mode if you have not yet set up your Sage Trac account.

(3) Pick a ticket, say #22148, to add to your Sage. The way we access tickets is by checking them out from trac,

```
[user@localhost:~/sage]$ git trac checkout 22148
```

When you do this some information will print out in the terminal including which files are part of the new branch.

(4) Take a peak at which branch you are in now

```
[user@localhost:~/sage]$ git branch
```

Look, it's one corresponding to the ticket you checked out.

(5) Do a quick Sage build with `make build`.

(6) If you checked out ticket #22148 a new function will be available to you `solve_S_unit_equation`. It inputs a number field, K, a ring of S-units, and a precision. It then returns the set of solutions to the equation $x + y = 1$ which lie in the S-units.

To learn about it try typing

```
sage:  ?solve_S_unit_equation
```

Notice that the ticket has not been added to Sage, but maybe some day.

(7) We use the same git steps as we did before, adding and commiting any changes we make to Trac tickets. There is an added step at the end of pushing the changes to Trac. You can attempt to push at any time, if no changes have been made, git will let you know.

```
[user@localhost:~/sage]$ git trac push
```

(8) If the ticket has been changed since your original checkout, there's an easy way to get the updates. Make sure you're on the correct branch using `git branch`. Then type

```
[user@localhost:∼/sage]$ git trac pull
```

4. The Fluff

Now that we know how to write code and add it to Sage, we have to be able to write the document strings, those nice helpful notes that come up when you put a '?' before a function in Sage.

(1) Switch back over to your `test` branch. We will do this there.
(2) Open `test.py` in your favorite text editor.
   (a) The first line is a description of your function. This should be short, sweet, and to the point.
   (b) Two lines later you place your inputs, as a list. Notice the double tick marks around the `n`. This will format it as code when the html help documents are made. Similarly the single tick mark will function the same as a `$`, formatting it as a LaTeX'd character. You could also use the `$` here if you'd prefer.
   (c) Third here is the output. We describe these and do not indicate what they were called within the function because the user doesn't care.
   (d) We end with examples. These should be testable. The first line in each example is the code one would type in and the second is the output one would receive if that's exactly what they did.

   These four items are required for every single function that is added into Sage. In addition there are several other sections we can put into doc strings, they are `REFERENCES`, `SEEALSO`, `ALGORITHM`, `NOTE`, `WARNING`, `TODO`, `PLOT`, and `TESTS`. If you want more specific information on these things, check out the developer guide:

`http://doc.sagemath.org/html/en/developer/coding_basics.html#documentation-strings`

(3) The format of the document strings is very important. Here is what it should look like for the Fibonacci function,

```
def Fibonacci(n):
    r"""
    Return the ''n''th Fibonacci number where the first and second are '1'

    INPUT:

    - ''n'' -- an integer

    OUTPUT:

    The integer that is the ''n''th FIbonacci number

    EXAMPLES::

        sage:  Fibonacci(5)
        5

    The function can be stacked

    ::

        sage:  Fibonacci(Fibonacci(7))
        233
    """
```

Go ahead and add exactly this to your `test.py` file, rebuild sage with `./sage -b > output`, open it up and type `?Fibonacci`. Look at your handiwork.

(4) Here are my tips: Spacing in very important. For example, the number of colons indicates how indented the next line should be. Notice the `INPUT` and `OUTPUT` have a single colon after them so there is no indentation on what follows. On the other hand, `EXAMPLES` is followed by two colons, there the examples are indented.

Now write out a doc string like this one here for the other function you wrote.

(5) We can then test that the examples were put in there correctly. Quit Sage and in your terminal type

```
[user@localhost:~/sage]$ ./sage -t src/sage/rings/number_field/test.py
```

Now this does not test everything it simply makes sure the examples all work verbatim.

Why don't you try changing the output of `sage:  Fibonacci(5)` to `5.0` and run a doc test to illustrate this point.

An important note when doc testing is that you rebuild Sage whenever you make changes to it! Otherwise you'll get errors that you no longer have.

(6) To check that all of your functions have doc strings you can type

```
[user@localhost:~/sage]$ ./sage -coverage src/sage/rings/number_field/test.py
```

5. Installing and Initializing git

(1) Download and install the appropriate version of git

`http://doc.sagemath.org/html/en/developer/git_setup.html`.

In fact you can get everything I'm about to say from that website, but this is just the bare-bones version.

(2) Once this is installed you need to configure your local git repository. Open a terminal window, and from the ∼ directory, type the following:

```
[user@localhost:~]$ git config --global user.name "Mckenzie West"
[user@localhost:~]$ git config --global user.email myemail@gmail.com
```

Except obviously replace my information with your own. You can check that this made its way to your configuration file `~/.gitconfig`.

6. Installing Sage using git

This method is used to install Sage on your own machine. To create your own instance of Sage on the k8s server, check out the Wiki under the title *Sage installations*.

(1) Begin by opening a terminal and navigating via `cd` to the directory in which you want your sage build to exist, this is typically the ∼ directory.

(2) After that, clone the repository by typing:

```
[user@localhost:~]$ git clone git://github.com/sagemath/sage.git
```

(3) Type `ls` into the terminal to see that a directory named `sage` has been added. Switch over to this directory using `cd sage`.

(4) If you are going to do development for Sage you want to be on the development branch. Use git to switch over:

```
[user@localhost:~/sage] git checkout develop
```

See the next section for more about branches and git.

(5) We then must compile Sage, this takes a LONG time, so if you already have a copy of Sage on your computer don't worry about this step. To compile, type

```
[user@localhost:~/sage]$ make
```

7. The Branches of git

Ok, you're here because you aren't sure what's going on with git. Using git allows us to make changes to documents, share them, forget them, etc. The way I think of it is that git is a lot like Dropbox but is way better for programming.

So the first thing to know about git is that it allows you to make changes to files without the fear of ruining everything. The way we do this is by creating our own branches. To start you may be in the `develop` branch of Sage. Either that or the `master` branch.

(1) You can determine which one of these it is by going to your terminal and typing

```
[user@localhost:sage]$ git branch
```

It will then give you some information that looks like

```
* develop
  master
```

The `*` indicates which branch you are in. Try it out yourself.

(2) Now maybe you want to try playing around with code. To avoid ruining anything you should create a new branch, say `test`. There are two ways to do this.

(a) Create the branch using

```
[user@localhost:sage]$ git branch test
```

Then move over to the branch by “checking it out”.

```
[user@localhost:sage]$ git checkout test
```

(b) The second is to do both simultaneously:

```
[user@localhost:sage]$ git checkout -b test
```

Either way works, so choose your favorite.

(3) Let’s return to the `develop` branch by simply typing `git checkout develop` command in the terminal. (If you started on `master` go ahead and return to that.)

(4) Now we can delete the `test` branch we made by typing

```
[user@localhost:sage]$ git branch -D test
```