Python Tutorial for the Sankey Lab

August 25, 2014

Contents

1 Installing everything

1.1 Windows 7

1. Install Python(x,y). It has everything. Almost.
2. Install pyqtgraph

1.2 Mac OSX

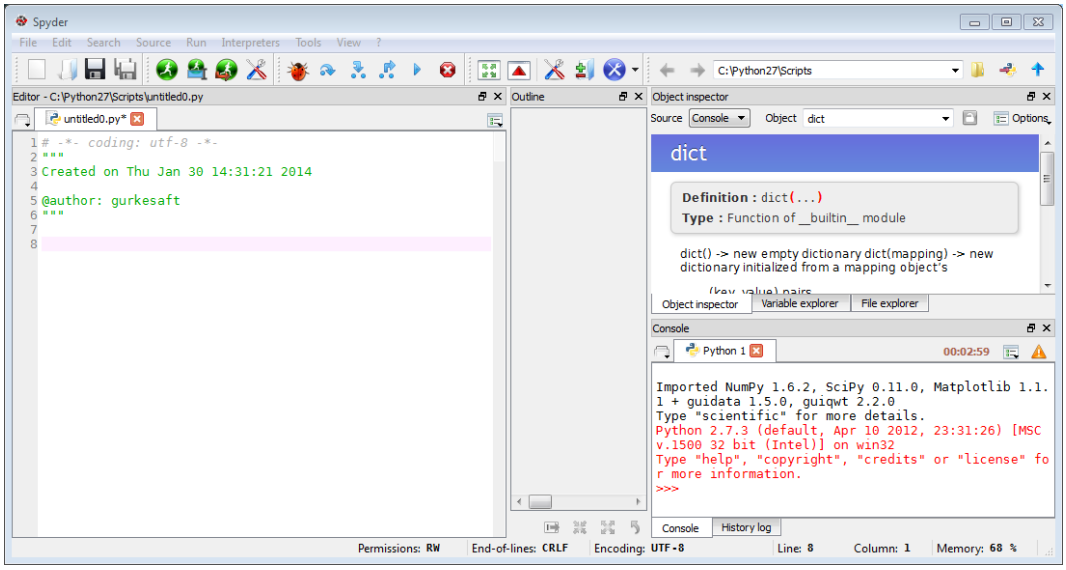
1. Install Homebrew
2. For python 2.7, use these commands in the terminal (for other pythons, modify appropriately):
   - $ brew install python

- $ brew install pyqt
- $ brew install gfortran
- $ brew install pkg-config
- $ easy_install pip
- $ pip-2.7 install numpy (note: on some macs this might be pip2.7)
- $ pip-2.7 install scipy
- $ pip-2.7 install matplotlib
- $ pip-2.7 install spyder (note: recently couldn't find this package on at least one mac, but spyderlib now has a dmg for osx)
- $ pip-2.7 install pyqtgraph

2 Spyder

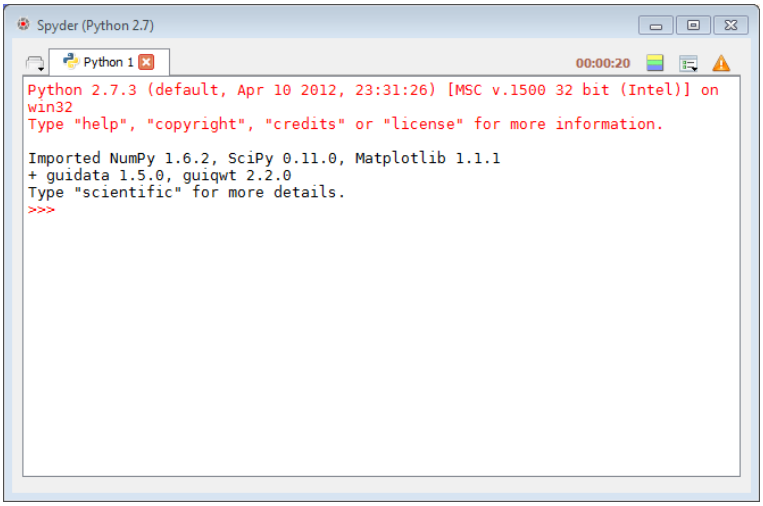
2.1 Getting Spyder up and running

1. First, open **Spyder**, a.k.a. "the most amazing coding environment I have ever seen". In Windows, this should be in your start menu, under Python(x,y). In OSX, this should be an application, or you can type "spyder" into the interpreter. When it launches (takes awhile), you should see something like this:



   On the left is a file editor, on the right is the "object inspector" (above) and the interpreter area (below). We will use the editor and interpreter (sometimes called the "python console") a *lot*.

2. Close the existing interpreter, right click the gray area, and select "Open a Python interpreter". This gives you a clean slate to work with. Note: you can also run **Spyder (light)** if you just want to play with a interpreter & no editor. I often do this when plotting so as to save screen space. Here is **Spyder (light)**:



For pure interpreter testing in this tutorial, you will sometimes see these images, but the functionality is the same as the interpreter embedded in the editor above.

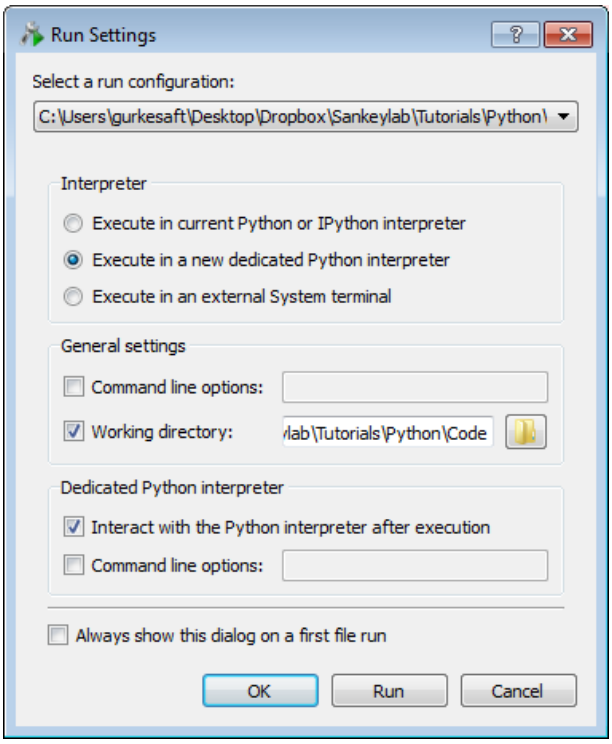
2.2 Writing a script and running it in the interpreter

Let's run our first python script:

1. In the editor, delete all the junk at the top of the file and write the following code:

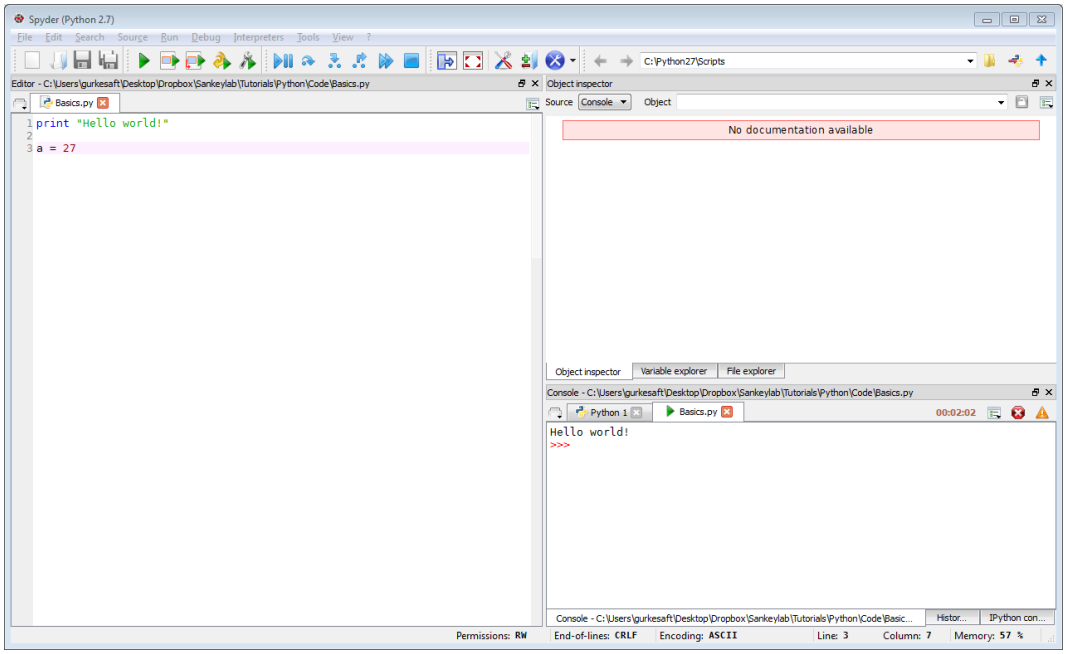
```
print "Hello world!"

a = 27
```

2. Select from the menu "File -> Save As..." and save the file into a convenient directory of your choosing (we will use this directory for all of our tutorial scripts).
3. Select from the menu "Run -> Configure...", and make sure it looks like the one below, and click "OK".



- **Execute in a new dedicated Python interpreter:** ensures that your script will run with a clean slate every time.
- **Working directory:** you will not have to modify this; it is the first directory Python checks whenever you specify a filename.
- **Interact with the Python interpreter after execution:** this allows us to play around and troubleshoot after the script has completed.

4. Run the script! Either press the "run file" button, type F5, or select from the menu "Run -> Run". You should see the following:



The three lines of code in the editor window have now been executed by a fresh, new python interpreter. The first line printed the **string** (section **??**) "Hello world!", the second (blank) did nothing, and the third, set the **variable** (section 3.1) named "a" to value 27. The "=" sign is used to assign values to variables. Note that this last command did not produce any visible result in the interpreter. However, it did do some stuff behind the scenes. If we now type

```
>>> a
```

into the interpreter and press enter, we see the value of a.

```
>>> a
27
```

furthermore, we can attempt some math using "a" (discussed more in section 3.1):

```
>>> a+7
34
```

we could also do all of the stuff in the editor window right in the command line, i.e.:

```
>>> print "Hello world!"
Hello world!
```

So, basically that's it. You write a bunch of stuff in a file (the "script") then run it in the interpreter. Whatever is defined in the script can be then monkeyed with in the interpreter.

3 Python Basics

3.1 Built-in math

Python is a calculator. You can add, subtract, multiply, exponentiate numbers, and divide as shown below, and python will respond as you might expect:

```
>>> 1+1
2
>>> 2*2
4
>>> 2**5
32
>>> 5/3
1
```

I intentionally wrote a weird result on the last command (try this!) to illustrate a common issue with those new to python: unless you somehow tell python you are working with non-integers, python will perform calculations using *integer math*, meaning $5/3 = 1$ remainder 2. To get the remainder, use the "mod" operation:

```
>>> 5%3
2
```

Why default to integer math? Because it's *fast*. If you want to stick to "floating point" math (i.e. "having a decimal point"), just add a decimal point in the right place. Whenever python combines integers with floating point numbers, the result is a floating point number. More examples:

```
>>> 3.0/2
1.5
>>> 2.0**0.5
1.4142135623730951
>>> (5+2.0)/2
3.5
```

Note, however, that this decision is made in the order of operations. So the third command above was computed as $(5+2.0)/2 = 7.0/2 = 3.5$, but

```
>>> (5+2.0)/(3/2)
7.0
```

Note also: this behavior depends on the shell you are using. As of 2014, the spyder shell will always assume floating point, and other python shells do not. Hence, it is best to *always* take this into consideration so that when you send someone your code it doesn't fall apart completely!

3.2 Variables

Like most programming languages, values can be store in **variables**, for later use, e.g.

```
>>> a = 5.0/2.0
>>> b = 3.0*a
>>> a+b
10.0
```

Good times.

3.3 Functions

Functions are objects that receive inputs, do things with those inputs, and make some outputs. Let's define one at the command line:

```
>>> def f(x,y): return x*y
```

Press enter twice after this to finish defining the function[1]. This function takes two inputs, x and y, multiplies them, and returns the result. Try it!

```
>>> f(32,2)
64
```

You can also do things with the result of a function directly, e.g.

```
>>> f(32,2)*2+1
129
```

This function is too simple, and super boring. To define a multiline function, let's make a script in spyder and add this code:

```
def f2(x,y,z):
    print "x =", x
    print "y =", y
    return x+y*z
```

When we run this script, we can now use this function from the command line:

```
>>> a = f2(1,2,3)
x = 1
y = 2
```

Here the function printed the x and y values we sent it, then returned x+y*z. We stored this result (7) in the variable "a".

Note that python is very strict about indentation. The "contents" of the function must all be indented by the same amount, and python stops defining the function when it sees that you have stopped indenting. Thus you can define several functions in one script (and run them in the same script if you like!):

```
def f2(x,y,z):
    print "f2 executed"
    return x+y*z

def f3(x):
        print "f3 executed"
    return x**x

print f3(2) + f2(1,2,3)
```

Try running this script and see if you understand the output.

3.4 Strings, Lists, Tuples, and Dictionaries

There are many objects in python other than numbers, too. I constantly use the following.

3.4.1 Strings

One of the most amazing things python can do is manipulate text. Text objects are called "strings" in python. Let's make two strings:

```
>>> s1 = "my first string"
>>> s2 = 'my "second" string'
```

Strings are defined by enclosing some text within either single or double quotes. There is no difference, though enclosing by single quotes allows you to use double quotes in the string itself, and *vis versa*. If you need to use both (or other funny characters like "line breaks", use "escape characters" which consist of a backslash "\" followed by a character. For example:

```
>>> s3 = "my \"third\" string \n has a line break."
```

[1] Note you can also make multiline functions at the command line. How this works is different from console to console.

The \" allows you to include the quotes without ending the string, and the \n adds a line break. Try inspecting s3:

```
>>> print s3
my "third" string
 has a line break.
>>>
```

Note the space on either side of \n is not necessary.

Strings also have some amazing functionality, for example you can add them:

```
>>> s1+s3
'my first stringmy "third" string\n has a line break.'
```

You can get a particular character (zero corresponds to the first):

```
>>> s1[3]
f
```

You can get a subset of characters:

```
>>> s1[3:8]
'first'
```

And you can split them:

```
>>> s1.split('s')
['my fir', 't ', 'tring']
```

- Note the difference between square brackets [] and parenthesis (). Parenthesis are used for calling functions, and square brackets are used for referencing data (in this case, the characters of the string).
- Note also we have used a dot "." to access built-in functionality of the string s1. This split() function splits the string (in this case by the delimiter 's') and returns a *list* (see next section) of three sub-strings, without affecting the original string s1. Such a function is referred to as a "method of an object" and is a nifty feature of "object-oriented coding". All python objects have their own built-in functionality, as well as the data they hold. We'll talk more about this later! For now, just know that if you have an object, you can see all of the things it can do by typing a "." after your variable name and looking at the list that pops up in spyder. Playtime ensues, and this is a great way to learn about objects!

3.4.2 Lists

A list is an object that can hold many other objects (including other lists). Let's create one.

```
>>> a = [3.4, 27, "test"]
```

Here the list is surrounded by square brackets, and each element is separated by a comma. This list has 3 elements: two numbers and a string. To access an element or range of elements,

```
>>> a[1]
27
>>> a[1:3]
[27, "test"]
```

These "functions" also return values and do not change the original list. Try typing "a." and looking for all of the list functionality. You can remove elements with a.pop(), add elements with a.append(), sort them, etc.

3.4.3 Tuples

Tuples are like lists but they are defined with parenthesis instead of brackets:

```
>>> a = (3.4, 27, "test")
>>> a[0]
3.4
```

You can read about the various differences between the two, but in practice for me, I don't use them unless I have to because they don't have as much functionality as lists. They often appear in the context of function definitions for me, e.g.:

```
>>> def f(*a): print a
```

Putting a * before the input specifies that you can call f() with as many arguments as you like, and they will be stored in a *tuple* named "a". So:

```
>>> f(1,2,3,4,"test")
(1, 2, 3, 4, "test", )
```

This adds some serious flexibility to function definitions!

3.4.4 Dictionaries

Dictionaries are one of the most powerful python objects I have found. As the name suggests, it's an object with which you can "look up" values. Dictionaries are defined with some odd-looking syntax involving curly braces:

```
>>> d = {"python":"a programming language", 32:"entry 32", 44:128}
```

This dictionary has 3 entries. Each entry has a "key" (i.e. the thing to the left of the colon) and a "value" (i.e. the thing to the right of the colon). To access a list of keys or values:

```
>>> d.keys()
['python', 32, 44]
>>> d.values()
['a programming language', 'entry 32', 128]
```

To get values and play with them:

```
>>> d["python"]
'a programming language'
>>> d[32]
'entry 32'
>>> d[44] + 1
129
```

And to add new entries or remove entries:

```
>>> d['new entry'] = 444
>>> d.pop('python')
'a programming language'
>>> d
{32: 'entry 32', 44: 128, 'new entry': 444}
```

So, now you have a few ways to store data.

3.5 Logic and Loops

3.5.1 "If" Statements

Often you want to do different things based on logic. For this, we use "if" statements. Here is an example:

```
>>> if 3 == 2: print "what??"
```

As with the inline function definitions above, you must push enter twice to make this work. Here python checks whether 3 is equal to 2, and if it is, it prints a confused sentence. Since python is logical, nothing will be printed. If we change this line to

```
>>> if 3 > 2: print "fine."
```

it will print "fine.", because 3 is actually greater than 2. Let's define the following test function:

```
def f(a):
    if a%2 == 0:
        print "it's even."
        print "nice work."
    elif a%3 == 0:
        print "it's a multiple of 3."
    else:
        print "I don't know."
```

This function first checks if the argument is even by "modding" it with 2. If it is even, it prints "it's even" and "nice work". The following line should be read "else if" or "otherwise, if". If it's *not* even, this line of code is called: it checks if it is a multiple of 3 and prints "it's a multiple of 3" if it is. You can have as many "elif" statements as you like. Finally, if none of the if's are satisfied, the "else" or "otherwise" code is called, and it prints "I don't know."

Note that just like a function definition, indentation for if, elif, and else statements is strictly obeyed.

3.5.2 "For" Loops

Often one needs to repeat an action many times, and for this I often use "for" loops. For example, say we have a long list of numbers and we need to print all of the even values:

```
for n in [0,1,2,3,4,5,6,7,8]:
    if n%2 == 0: print n
```

Note that this is just iterating over the supplied list, and any list will do. For example, this can be written more succinctly as

```
for n in range(10):
    if n%2 == 0: print n
```

The "range()" function returns a list, which is iterable.

3.5.3 "While" Loops

Sometimes you want to loop until some condition is met, but you do not know ahead of time how many steps this will take. For this we use a "while" loop. For example, we might want to find the first prime number above 1000. Here we first define a (totally inefficient) function to determine whether a number is prime, then perform a "while" loop that tests numbers above 1000.

```
def is_prime(x):
    for i in range(2,x):
        if x%i == 0:
            print x, "has factor", i
            return False
    return True

n = 1000
while not is_prime(n):
    n = n+1

print "First prime above 1000 =", n
```

This code results in the following output:

```
1000 has factor 2
1001 has factor 7
1002 has factor 2
1003 has factor 17
1004 has factor 2
1005 has factor 3
1006 has factor 2
1007 has factor 19
```

```
1008 has factor 2
First prime above 1000 = 1009
```

I did not know that. Thank you, python.

3.6 Comments (a.k.a. the most important things you'll ever finally learn to use a few years from now)

Until now we have only written code. The other thing we can (MUST) include in scripts is ***comments***. A comment is a portion of the script that python will ignore. They are used primarily to give information to the human being that is reading your code. This includes other lab members and your future self. I guarantee that no matter what I say, two things will happen to you:

1. You will write code without comments, because what you are doing is so simple "it doesn't need comments".
2. You will return to your code N months later, have no idea what the code is doing, and lose several days of progress trying to add one small feature.

This is a promise, and you will find it happening even for $N \sim \pi$. I have yet to find a way to sufficiently emphasize the importance of comments so as to avoid these events and for this I am sorry.

There are several ways to add comments in python. The simplest is to type the pound sign "#" before some code, e.g.

```
# some notes about what is about to happen
print "some stuff" # some notes about this particular line
```

What this will do is simply print "some stuff". Everything after the # signs on each line is ignored by python. You can also use the # symbol to temporarily disable some code without deleting it, and in spyder, you can select a big block of code and comment all of it by typing ctrl-1.

Another way to add comments is to place some code between triple quotes, either

```
"""
Some stuff python will ignore.
You can write a few lines

or even some code:
x = 32
"""
print "some stuff."
```

Everything but the print statement will be ignored. Typically, the "three quote" comment is used below a function definition, such as:

```
def f(x):
    """
    You can describe what the function does here. In
    this case, it returns the square of x.
    """
    return x*x
```

The great part about this type of commenting is that the information in between the triple quotes will appears as the documentation of the function, either above the function as you are typing it or when you type help(f).

For example, we should comment the "while loop" code above:

```
def is_prime(x):
    """
    Function that (in the silliest way possible)
    determines whether x is prime.
    """

    # check if it is not prime
    for i in range(2,x):
```

```
        # if there is no remainder, i is
        # not a factor of x
        if x%i == 0:

            # print the naughty factor
            print x, "has factor", i

            return False

    # no factors found!
    return True

# look for the first prime number above 1000
n = 1000
while not is_prime(n):
    n = n+1

print "First prime above 1000 =", n
```

Much easier to read!

Advice: write the comments describing what is about to happen before writing the code. You will spend a lot more time debugging code than writing code, so having clear, natural-language blocks of text will greatly simplify your experience. I typically try to write more comments than code, because it simply does not add a any time to the coding process compared to learning how the actual code works or looking up functions online, etc. It also will save you factors of 10 in debugging time.

You'll see.

A few years from now, anyway.

3.7 Modules

Python's built-in functionality (some of which is described above) can be extended *dramatically* with "modules". For example, a common module for very fast numerical calculations, people typically use the "numpy" module, which can be "imported" (after you install it, of course!) as follows:

```
>>> import numpy
```

The module itself is an object, and after you run the above command, you can access all the module's functionality with the dot ".", for example:

```
>>> numpy.linspace(0,10,3)
array([  0.,   5.,  10.])
```

In this case, we used the linspace() function to create a 3-element *array* (discussed below) spanning the range 0 to 10. There are a lot of modules. We can also write our own modules. If you don't like typing "numpy" all the time, you can shorten it:

```
>>> import numpy as np
>>> np.linspace(0, np.pi, 5)
array([ 0.        ,  0.78539816,  1.57079633,  2.35619449,  3.14159265])
```

Or you can import everything into your "name space" so that you have access to it directly:

```
>>> from numpy import *
>>> linspace(0, pi, 5)
array([ 0.        ,  0.78539816,  1.57079633,  2.35619449,  3.14159265])
```

I'm not a fan because it clutters the name space, making any modules I write difficult to navigate with the pop-up code completion.

All of python's modules are located in a single "site-packages" ("dist-packages" on linux) folder. To find the location of this folder on your computer, use the following commands:

```
>>> import sys
>>> for p in sys.path: print p
...
C:\Python27\lib\site-packages\spyderlib\utils\external
C:\Python27
C:\Python27\Scripts\
C:\Python27\Lib\site-packages
...
```

Note you'll have to push enter twice after the for loop. My actual list is larger than the one shown here, but you can quickly see where "site-packages" lives on my computer. Once you find this, remember it forever, because if you ever have to manually install a python package (such as spinmob, described below), you will need to have access to this folder.

3.7.1 The numpy module

Python (like Matlab) is an interpreted language, meaning it is not compiled into fast machine-level code before executing, meaning it is incredibly slow. The solution is to perform calculations using "numpy arrays" (like those above), which *are* compiled into fast machine-level code. For example, if I want to numerically integrate the function $\int_0^1 \sin(x)dx$, I *could* do this:

```
import numpy as np

# create an array of a million x-values between 0 and 1
xs = np.arange(0.0, 1.0, 1e-6)

# use python to loop over each value and add it to the sum:

# start with sum = 0
sum = 0

# loop over each element of the array
for x in xs:

    # add this value times dx to the total sum
    sum = sum + np.sin(x) * 1e-6

print sum
```

This works, and it is correct. However, on my laptop, this takes about 10 seconds, because it takes time for python to interpret what amounts to a million lines of code! If instead I do the following:

```
sum = np.sum(np.sin(xs)) * 1e-6
```

I get the answer *immediately* because the underlying numpy code is blindingly fast. Note that when I type np.sin(xs), numpy loops over the entire array, taking sin() of each element, returning an array of the same size. The command np.sum() then tell numpy to add all the elements together.

In short, anything that can be done to an individual number in python can also be done to a numpy array of any size or shape. See the numpy documentation for more details!

3.7.2 The scipy module

This is the other biggy. In this module you will find fast-compiled code for solving differential equations, taking smarter integrals than the one above, fitting, and a whole lot more. Definitely look into this module. Basically every well-established mathematical trick you can think of has already been created for you.

4 Git, Github & Spinmob

Spinmob is a homemade library with for saving, loading, plotting, fitting, and otherwise analyzing data. It now also includes the "Easy GUI Generator" library we use for taking data. More information (including a growing

wiki-based tutorial!) can be found at github:

https://github.com/Spinmob/spinmob

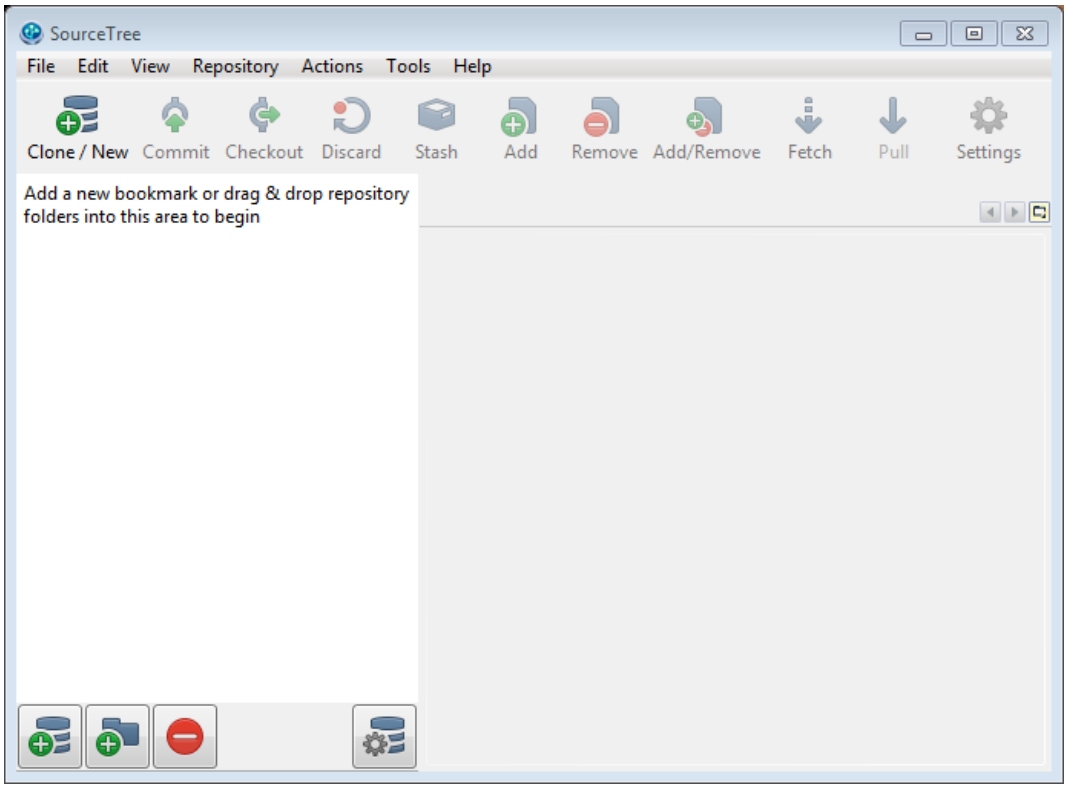
To install the spinmob module, you can simply download the source into a (new) folder site-packages/spinmob/, but it is recommended instead to install a "git client" and use this to perform a "git clone" into this directory. Using git will allow you to get the latest version of the code, instantaneously switch between different versions of the code, create a new branch, modify the code, and / or submit updates / fixes to me via "git pull requests".

4.1 Git Client

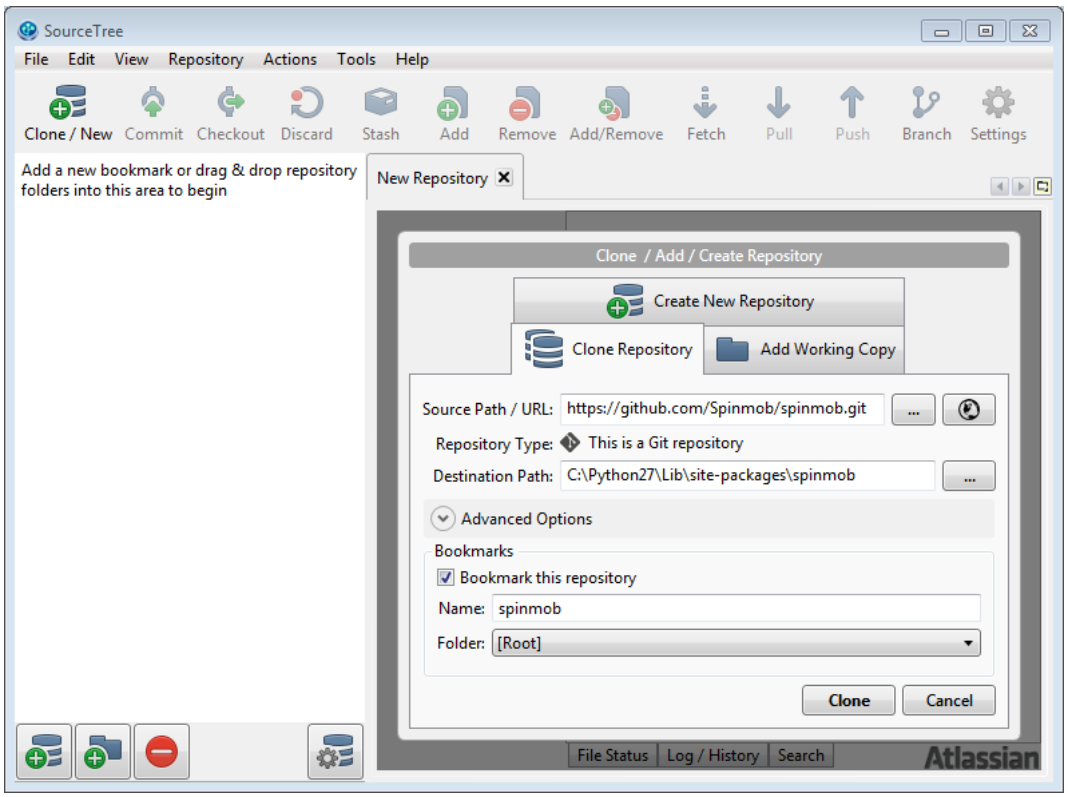
Though *somewhat* clunky, my current favorite no-nonsense git client is "SourceTree", which can be freely downloaded here:

http://www.sourcetreeapp.com/

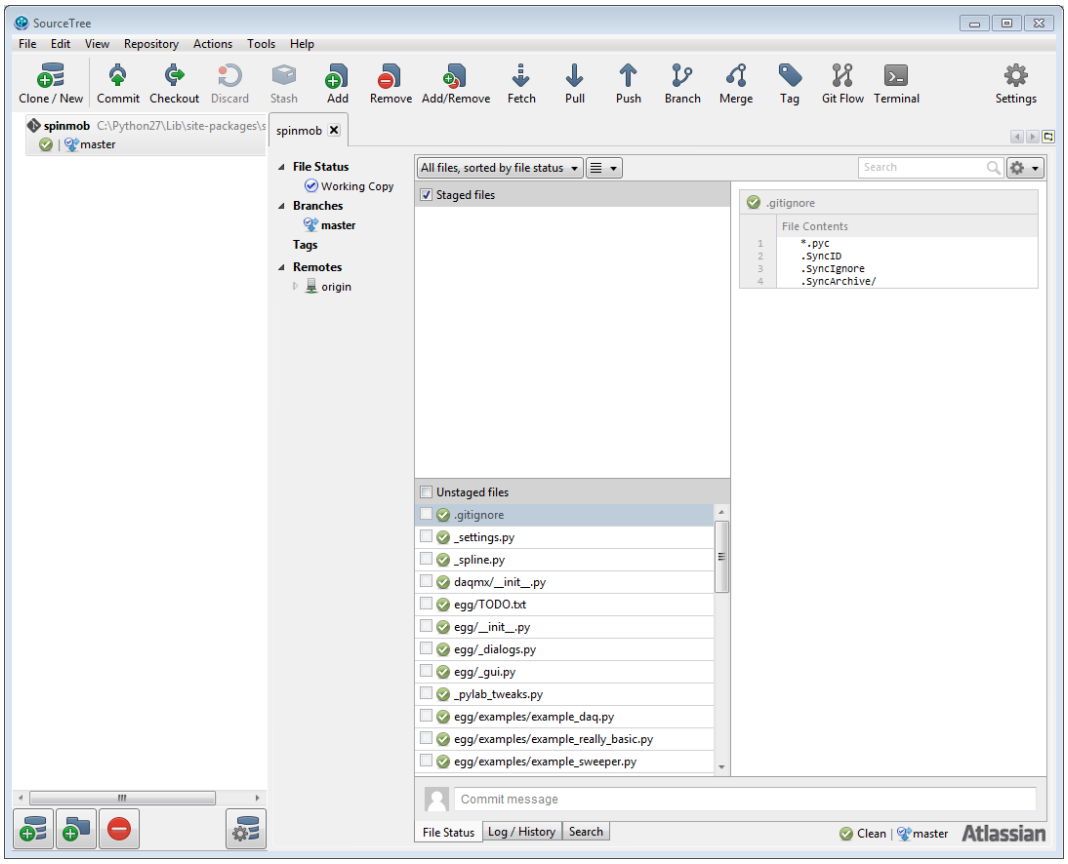
After installing and starting the program, you should see something like this:



Press the "Clone / New" button to begin adding a repository:

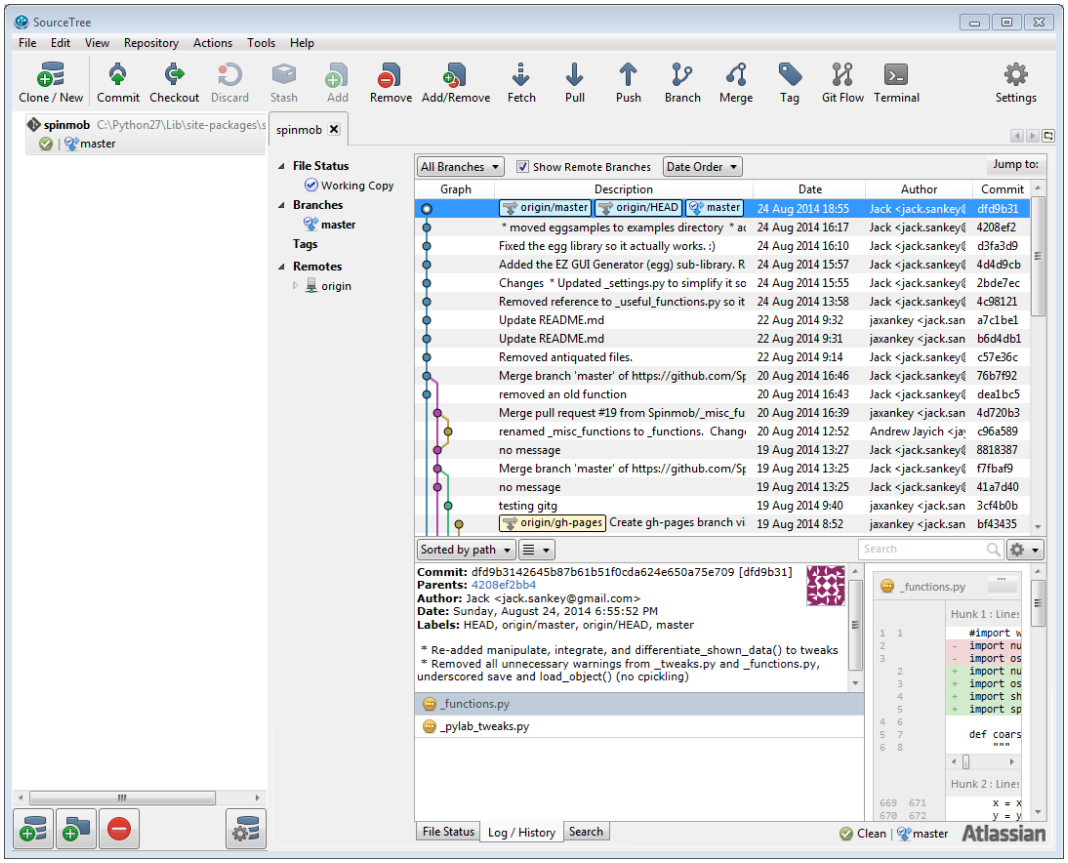


The location of the spinmob "Source Path / URL" can be found on the spinmob github site mentioned above. The "Destination Path" should be a (new) "spinmob" directory in python's "site-packages" directory (see section 3.7). When the information is entered, press "Clone", and it should download the latest files from the online "master" branch. If you then click "Working Copy", you should see all the files:



Note you may have to select "All Files" from the pull-down menu to see them all. If you modify files, SourceTree will notice, and mark them in this area. From here you can create your own branches, make your own modifications to the source locally, and revert to the master branch / previous versions if you mess something up. None of this will affect the "master" branch, which is moderated by me.

If you click the "master" branch, you can see all of the branches and history of spinmob, back to the day it was first uploaded. From here you can switch versions or update to the latest:



Right-clicking any of the versions will bring up the option to "Checkout...", which means "switch to this version". This is useful for updating to the latest version and (if an update breaks part of the code) roll back to a previous version without any headache.

If you learn more about python, spinmob, and git, I will definitely welcome "pull requests" for any fun code you would like to contribute to spinmob. Let me know if you make / fix / enhance something!