

NYU PHYSICS

DataMaster Manual

Patrick Anker

Kaitlyn Morrell

Contents

Why DataMaster?	1
Requirements & Setup	2
Usage	3
Code Structure	4
Commands	5
Example	8
Potential Problems	13
References	15

Why DataMaster?

Most of the analysis codework that you do in the start of your undergraduate career – if any – ultimately comes down to simple tasks that you can do on your own. However, physics is a cooperative science and so is its data analysis. Eventually you will come to find that the analysis work for labs outweighs the time spent in the lab itself, which means that you may be spending many hours staring at some sort of text editor. You may only have a weak understanding of some Python, which makes the analysis codework even scarier. This is why it's highly, highly, *highly* recommended to work with others in the lab *and* on the analysis; it's much better to bounce analysis ideas off of each other than it is to let them stew and get muddled in your own head.

Working together on lab analysis can be cumbersome, if you do it naively. When we first started working together in Intermediate Lab II, we wrote the analysis code together using Google Drive and a single executable Python script. It was a nightmare: we were going back and forth, commenting out plot commands for plots that we didn't want and doing our best to convince the Drive server to sync. We failed on both counts.

For our second lab we wanted to resolve these issues: 1) make it easier to work on code together and 2) make some sort of project manager which handles all of the plot and print commands. We used [GitHub](#) to solve the first problem – though I imagine any good version control software that is *not* Google Drive (or its ilk) would solve that issue. The solution to the second problem was DataMaster.

DataMaster is a Python script manager that allows for control over which output from the lab analysis code is being viewed. It makes an – at times – laborious process exponentially less tedious than it could be. For this reason alone, we highly recommend using DataMaster for your lab analysis.

Requirements & Setup

DataMaster has been tested on and is stable with Python versions **2.7** and **3.6**. If you have [Jupyter](#) or some other [IPython](#) implementation, you will most likely already have the [SciPy](#) stack; that is, provided you have these packages, you should already have the requirements for DataMaster to run. Specifically, you need these packages for DataMaster to run:

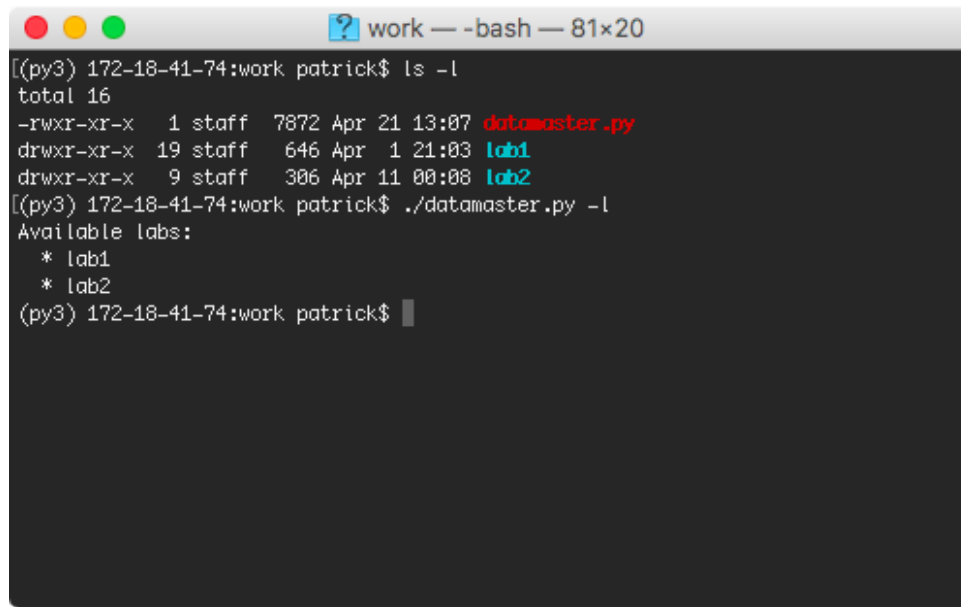
1. Python 2 or 3
2. Matplotlib

With the basic requirements out of the way, we need to talk about the file structure for each DataMaster project – whether that be the labs for a semester or some research project. The DataMaster script itself can go into any directory (folder) of your choice; however, DataMaster uses the child directories of DataMaster's directory as lab names – like so:

```
~/some/working/directory/  
  datamaster.py  
  lab1/  
    __init__.py  
    lab.py  
  lab2/  
    __init__.py  
    lab.py
```

Indeed, DataMaster uses the names of the child directories themselves as the names of the labs, as shown in Fig. 1.

As you can see, all of the lab directories have two files: “**lab.py**” and “**__init__.py**”. “lab.py” is where all of the analysis code will be. “__init__.py” is an artifact of the Python language itself, which sadly must be in all lab directories. Thankfully, all “__init__.py” documents have these exact lines:

A terminal window titled "work — -bash — 81x20" shows the following commands and output:

```
[(py3) 172-18-41-74:work patrick$ ls -l
total 16
-rwxr-xr-x  1 staff  7872 Apr 21 13:07 datamaster.py
drwxr-xr-x 19 staff   646 Apr  1 21:03 lab1
drwxr-xr-x  9 staff   306 Apr 11 00:08 lab2
[(py3) 172-18-41-74:work patrick$ ./datamaster.py -l
Available labs:
 * lab1
 * lab2
(py3) 172-18-41-74:work patrick$
```

Figure 1: Note how DataMaster’s lab listing matches the names of the directories in the current working directory.

```
# some-lab/__init__.py
from . import lab

__all__ = ('lab')
```

The first line is just a comment indicating that the above lines are in “__init__.py” of the lab “some-lab.” The second line “`from . import lab`” tells the Python module system to look through the current (lab) directory and find all documents with the filename “lab.py.” *There should only be one file named “lab.py.” per directory, otherwise DataMaster will not work!* The third line “`__all__ = ('lab')`” tells Python that the only relevant component of this directory’s module is the lab analysis document. Technically, this is the magic line that allows a module to be imported via the “`import`” command in Python.

These “**lab.py**” and “**__init__.py**” files are the only files that you need to include in the directories corresponding to different labs for DataMaster to work. The “**__init__.py**” will be the same for every lab, while the “**lab.py**” will contain the lab analysis code from which specific commands can be called as detailed in the next section.

Usage

DataMaster itself is a script that handles your data output with a console environment, but there are necessary structures that you need to have in your code that make it readable for DataMaster.

For this reason, there are two “levels” of access to the DataMaster platform: the code structure and the commands in the terminal environment.

Code Structure

The “**lab.py**” document is the heart and soul of your lab analysis. The whole point of DataMaster is to allow you to focus on the analysis itself and not the display overhead. When you load a lab, DataMaster searches for two sets of functions:

Function Name	Description
<code>get_something()</code>	Tells DataMaster that this function returns a value that will be displayed in the terminal
<code>plot_something()</code>	Tells DataMaster that inside of this function is a plot – using Matplotlib – that you want to generate
<code>run_something()</code>	Tells DataMaster that this function is a void returned function; useful for preparing large chunks of data in memory

The code structure for these functions should follow this setup:

```
def get_value():
    # Some number crunching, probably

    return value

def plot_thing():
    # Some data setup here perhaps..

    fig, ax = plt.subplots()
    # Code for the stuff you wish to plot

def run_runnable():
    # Code for this function (without a return statement)
```

As shown, all “get_” functions have a return value. “plot_” functions have some sort of figure setup – with “`plt.figure()`” or “`plt.subplots()`” – but they do not have “`plt.show()`.” DataMaster handles that for you.

The “run_” functions have **no** return value as they fall under a class of functions called “runnables.” If you know a bit of Java, you’ll know that these are actually multithreaded processes, i.e. they do not run at the same time as the calling script and use a separate chunk of CPU. *This is not the case for these functions!* DataMaster, for the time being, is a synchronous script with no built-in threading support. For larger projects, it may be useful to take advantage of asynchronous development, but for most data crunching the synchronous architecture is preferred. The “run_” functions, in this case, act like “fire and forget” functions; for example, say you’re performing some signal processing and have 100 terms in your Fourier series. Instead of building each term over and over, you could build them all and store them in memory with a “run_” function.

You can access these functions in DataMaster itself using the “-g value,” “-p thing,” and “-x function” commands, which are explained below.

For advanced usage, there also exists a “terminate()” function which acts like a destructor function in languages like C++. Once a lab is called to reload, if “terminate()” is declared in the lab analysis document, it will be run before the lab reference is reloaded. For example, say there is a large cache dictionary called, conveniently, “CACHE” which should be cleared upon reload. If you have an object like this, it **must** be properly dereferenced otherwise there may be memory leaks!

```
# Empty cache dictionary which gets filled over time
CACHE = {}

# Properly dereferences any object in a cache tree so that the
# garbage collector can free memory
def purge_tree(d):
    for k, v in d.items():
        if isinstance(d[k], dict):
            purge_tree(d[k])
        else:
            d[k] = None

# Terminate command called by DataMaster itself
def terminate():

    # Since CACHE is a global variable in this context, the global
    # keyword must be used to specify that we're modifying the global
    global CACHE
    purge_tree(CACHE)
    CACHE = None
```

After marking each reference in the cache tree as “None,” Python treats the references as memory that’s ready to be freed. While the lab reloads with “-r” DataMaster calls the garbage collector to run just before the lab is reread from file. If you have a project which manipulates lots of data, it’s *highly* recommended to take advantage of this “terminate()” function.

Commands

There are two methods to use DataMaster: single-line command executions, and its built-in command line interface. However, worry not! The commands for DataMaster are exactly the same in both circumstances.

For those of you who have used a command line before, DataMaster has UNIX-like syntax for its command structure. For those of you who have not dealt with command line interfaces before, the previous statement was probably gibberish. In most cases, UNIX-like commands have this structure in a terminal:

```
program -c argument
```

For example, printing the help listing for DataMaster looks like this:

```
./datamaster.py -h
```

The “./datamaster.py” tells the terminal environment that it wants to run a script named “datamaster.py.” The “-h” is an argument passed to the script. The dash before the ‘c’ is a style thing; this is what we mean by UNIX-like syntax.

These are the commands available in DataMaster:

Command	Syntax	Description
-h, --help	-h	Prints out all the available commands in DataMaster as well as the version being used
-l, --list	-l	Prints out all the available labs in the current working directory. Furthermore, if a lab is selected, it will also print out all the available “get” and “plot” functions in the selected lab as well as indicate which lab is selected.
-s, --select	-s argument	Selects the lab with name “argument” you wish to work with
-r, --reload	-r	Reloads the currently selected lab
-g, --get	-g argument	Prints something in the terminal with the name “argument”
-p, --plot	-p argument	Displays a plot named “argument”
-x, --run	-x argument	Runs a custom function named “argument”
-e, --exit	-e	Exits the command-line interface

Command-Line Interface

In most cases, you will be using the command-line interface for DataMaster, which is the interactive part of the tool. The [example](#) below details the workflow for using the command-line interface. It's very straightforward to access the CLI. Just run this:

```
python datamaster.py
```

Running the script without any arguments will send it straight into the CLI mode!

Single-Line Commands

There are some situations where you may be only interested in one piece of data from your analysis code, e.g. you forgot the uncertainty for a certain calculated value and do not want use the command-line interface. DataMaster allows you to do this with the following syntax:

```
python datamaster.py -s lab_name [-g, -p] value
```

In order to access the lab in question, you must type the select command **before** using the data access commands (-g and -p). DataMaster interprets each command one-by-one, so if you run the data access commands before the select command, it won't know from which lab you wish to "get_" or "plot_."

Example

All example documents, including DataMaster itself, can be found on [its GitHub page](#). In particular, this example is in the aptly named "example" folder on that webpage.

As usual, an example of how to use our software is much more illuminating than a raw manual on its own. In this case, we wish to measure the resistance across a resistor. We know from Ohm's Law that $V = IR$.

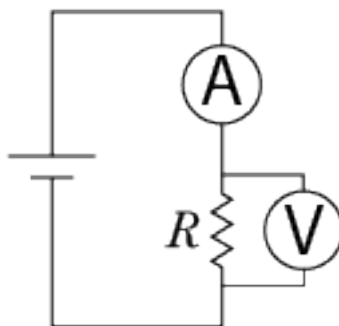


Figure 2: The circuit we are examining.

Our circuit, in Fig. 2, has an ammeter in series before a resistor (with resistance R) connected to some DC voltage source. Across the resistor is a voltmeter. For simplicity's sake, let's assume that both the ammeter and voltmeter are Fluke multimeters, standards tools in the undergrad labs. For each measurement, we select a voltage on the source, record the current supplied on the ammeter, and record the voltage drop across the resistor on the voltmeter

To begin our lab.py document, we need to import the necessary libraries for our analysis:

DC Voltage (V)	DC Current (mA)
0.503 ± 0.002	5.372 ± 0.001
1.043 ± 0.001	10.024 ± 0.002
1.526 ± 0.003	14.975 ± 0.002
2.034 ± 0.001	20.482 ± 0.001
2.521 ± 0.001	24.878 ± 0.001
3.018 ± 0.002	30.105 ± 0.003

Figure 3: Raw experimental data

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import curve_fit
```

You're probably already familiar with NumPy and Matplotlib, but you may not know about [curve_fit](#). This function is used in the linear regression function provided with the example document. Basically, it's a cleaner way to find regressions than what's detailed in Taylor [1]. The function in question is located [here](#) if you want to see the source code. For now, we'll treat it as a black box that builds a line as it's not necessary to understand for this example.

The raw data can just be put into NumPy arrays:

```
current = np.array([5.372, 10.024, 14.975, 20.482, 24.878, 30.105])
            * 1e-3 # mA
voltage = np.array([0.503, 1.043, 1.526, 2.034, 2.521, 3.018]) # V

# The multimeter tends to have a variable uncertainty, so these
# arrays is needed
dI      = np.array([0.001, 0.002, 0.002, 0.001, 0.001, 0.003]) * 1e
-3
dV      = np.array([0.002, 0.001, 0.003, 0.001, 0.001, 0.002])
```

Most of this work should be a familiar procedure by now, especially if you've had a semester or more of lab courses in the NYU undergrad program.

The next functions are explicitly for DataMaster:

```
def plot_line():

    # Least-squares linear regression for y = mx + b
    m, b, sy, sm, sb, r = lsq(current * 1e3, voltage) # We want to
    plot in mA
```

```

# You will NEED to call this for each plot so that you don't have
# multiple plots
# overlaid on each other
plt.figure()

# Range upon which we want to plot the line
x = np.linspace(5, 31, 1000)
plt.plot(x, m*x + b, 'c--')

plt.errorbar(x=(current * 1e3), y=voltage, xerr=(dI * 1e3), yerr=
    dV, fmt='r.', ecolor='k', alpha=0.5)

plt.xlabel('Current_($mA$)')
plt.ylabel('Voltage_($V$)')

def get_resistance():
    m, b, sy, sm, sb, r = lsq(current, voltage)

    # Resistance is the slope m; its uncertainty sm is already
    # computed by lsq()
    return (m, sm)

```

The `plot_line` and `get_resistance` functions will be loaded into DataMaster. Two things are important about the plotting function:

1. There is no need to include `plt.show()`. DataMaster handles this for you.
2. For each `plot_` function you write, you **must** include `plt.figure()` (or some other figure generating function like `plt.subplots()`). Otherwise, all the plots will display as a jumble in one figure.

`get_` functions can have any sort of return value, but they **must** have some return value. This can be a string or, in this example's case, an array with the resistance and the uncertainty in the resistance.

Accessing these functions' outputs is where DataMaster comes into play. When working on the lab analysis – that is, when you're simultaneously writing some code and checking to see if your code looks like what it's supposed to do via some plot or returned value – it is much easier to have a single running instance of DataMaster. This is why the “command-line interface” of DataMaster exists.

To begin, open up a command prompt and navigate to the directory where your copy of DataMaster is installed. For Windows and UNIX-like systems (OSX, Linux, FreeBSD, etc.), these navigation commands will look a little different:

```

# UNIX-like
cd ~/path/to/lab/documents/

# Windows

```

```
CD C:\path\to\lab\documents\
```

The ~ for the UNIX-like operating systems is shorthand for “/Users/user/” (OSX) or “/home/” (others), which is most likely where you keep your documents, e.g. the Desktop on OSX is “~/Desktop/.” Regardless, once you are at the correct working directory, the command to start DataMaster is the same for all operating systems:¹

```
python datamaster.py
```

After a couple seconds or so when you hit enter, a “less-than” symbol (>) should pop up. When you first run DataMaster, you should type “-h” to bring up the entire help listing for DataMaster. This is what you should see:

```
> -h
DataMaster version 1.1.0
Usage: datamaster.py -s <name> [-g, -p] <data name>

Commands:
  -h, --help: Prints out this help section
  -l, --list: Lists all the available labs and, if a lab is
              selected, all available gets and plots
  -s, --select <name>: Selects lab to compute data from
  -r, --reload: Reloads the selected lab from file
  -p, --plot <variable>: Calls a plotting function of the form "
                        plot_<variable>"
  -g, --get <variable>: Prints out a value from function of the
                        form "get_<variable>"
  -x, --run <variable>: Runs a custom function of the form "run_<
                        variable>"
  -e, --exit: Explicit command to exit from DataMaster CLI
>
```

We want to check to see if the example lab that we’ve been working is available to be used in DataMaster. For this, we should list all the available labs with “-l” which will print out something like this:

```
> -l
Available labs:
  * example
```

¹For those of you using OSX, you get the advantage of being able to run DataMaster *without* the python command. If you just type “./datamaster.py” in correct directory, the operating system will run the script automatically. If the terminal returns an error when you run this, it means the script’s permissions aren’t configured to be treated as “runnable.” To fix this, run “chmod 755 datamaster.py” in the terminal, which will correct the file’s permissions. Then, you’ll be set!

```
>
```

Perfect. The “example” lab is ready to be used. Now, select it with “-s example.” After a couple of seconds, DataMaster should print out this:

```
> -s example
Selected Lab: example

>
```

We are now set to run our data analysis displays, whether they be plots or just some numbers or strings. To see all the available functions we have access to, we use the list command “-l” again. Since DataMaster has selected a lab, it will now print the available functions in the selected lab “example,” as well as the other labs available:

```
> -l
Available labs:
> * example
-----
Functions for 'example'
Gets:
  * resistance
Plots:
  * line
>
```

Note that there is a ‘>’ in front of “example” now. As you perform more and more labs, you will have more and more available projects listed. To quickly indicate which one is the selected one, DataMaster will indicate it with a ‘>’ in the list command.

Since we know from the list above that the functions were correctly loaded into DataMaster, let’s load them up with “-g resistance” and “-p line” :

```
> -g resistance
(100.75994748210289, 1.5437565737907208)
> -p line
>
```

As you can see, the “get_” function returns the resistance and uncertainty straight into the terminal window. The plot, however, will create a whole new window *which will hang the DataMaster process until the plot window is closed*. The plotting function in PyPlot itself is known as a “blocking” function, which halts any code from running until its own process is terminated (by closing the plot windows).

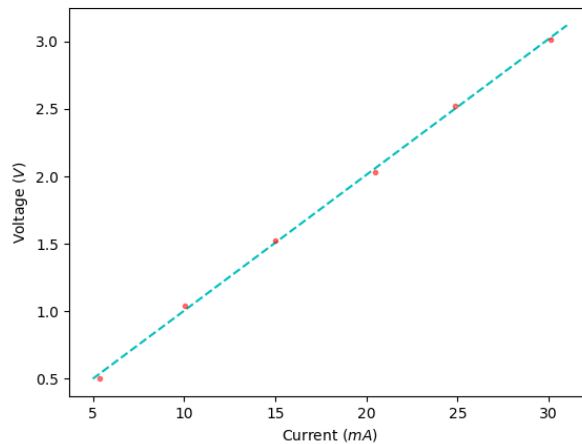


Figure 4: The data is pretty linearly correlated!

Note that it's also possible to get the plot and the resistance data at the same time by putting the commands all into one line:

```
> -g resistance -p line
```

You can string multiple commands into a single line and, as long as they are properly formatted, DataMaster will be able to interpret them all.

This is the general workflow for DataMaster: write some code and check to see if the desired outputs are correct. Many times, however, you will make mistakes: typos, sign errors, and so on. We all do this; we're only human. Even more, say you decided to make some other data outputs altogether, whether they be "get_" or "plot_" displays. Instead of quitting DataMaster and reloading the script to load up the changes, you need only reload your lab document with the reload command "-r" :

```
> -r
Selected Lab: example

>
```

The reload command will reload DataMaster's selected lab. If, however, there's some error in your analysis code that breaks the Python interpreter, DataMaster will remove the selected lab from memory as to not cause crashes. If this happens, you will need to use the select command "-s" again to load the file once the errors have been corrected.

Once you're satisfied with your work, it's best to quit and enjoy the rest of your day! There are two ways to safely exit the program: using exit command "-e" or using the built-in SIGKILL shortcut (in all operating systems that we know of) by hitting "CTRL-C" twice.

Potential Problems

There are some issues that may arise when working with DataMaster, some of which are actual bugs and some of which are issues with Python itself. These are just a few that we have ran across.

All of my plots are all jumbled together in one display. What happened?

If you work with Jupyter regularly, you may have developed the habit of dropping the “plt.figure()” or “plt.subplots()” command in your code. While Jupyter will take care of these commands for each of its own cells, PyPlot on its own cannot determine which plotting commands go to which figures. This is why it’s necessary to use either “plt.figure()” or “plt.subplots(),” depending on the situation. As such, for each “plot_” you write, you must include one of these commands.

I get an error when I try to select my lab. What’s happening?

There are two possibilities: you have an error in your Python code itself or the “__init__.py” file is either not present or correctly formatted:

```
> -s analysis
__init__ file not properly configured.
Could not load 'analysis'
```

If the above error is the one that you are getting, you must double-check to see that the “__init__.py” document exists in the same folder as the “lab.py” document **and** the init file’s contents match those in the [Requirements & Setup](#) section.

After I save a plot, DataMaster becomes unresponsive, and then I have to close the terminal window. What’s wrong?

This issue is strictly for OSX users. There is a bug in the windowing system that Matplotlib uses – called “tkinter” – that crashes when it has to interact with the OSX file system, for some reason. It’s annoying, but there’s a way around it. Instead of using “python datamaster.py” to start the script, use this:

```
pythonw datamaster.py
```

This “pythonw” hooks into the native OSX windowing system and prevents this crash happening *most of the time*. This is why OSX users can run “./datamaster.py” on its own in the terminal window. At the top of the DataMaster script, there is this line:

```
#!/usr/bin/env pythonw
```

This tells the terminal to use the “pythonw” environment to run this script. So, if you start DataMaster using this command, you should be set. There are some strange cases when, after extended periods of use with DataMaster, the process hang shows up again after saving a figure. In this case, the only way to resolve it is to close the terminal window.

References

- [1] John R. Taylor. *An Introduction to Error Analysis*. University Science Books, 1997. ISBN: 9780935702422.