gcmstools Documentation

Release 0.1.0

Ryan Nelson

January 13, 2015

CONTENTS

Welcome to the *gcmstools* documentation.

CHAPTER

ONE

GETTING STARTED

gcmstools is a Python package that reads some GCMS file formats and does simple fitting. The source code for this project can be found on GitHub. If you are reading this in PDF format, there is online documentation as well.

This user guide is broken into a few sections.

1. *Installation*: Information about getting a Python installation up and running and installing *gcmstools*.
2. *Basic Usage*: This section covers the usage of *gcmstools* to manipulate and plot GCMS data.
3. *Fitting*: The information in this section covers manual fitting of a GCMS data set.
4. *Calibration* and *Batch Processing*: These final sections cover calibration curves and automated data extraction.

You can skip to the final sections if all you want to do is automate some data extractions. It is not necessary to understand the basics of data manipulation/plotting as those tasks are automated in the calibration and batch processing steps.

Note: The examples presented in this document require a basic working knowledge of a command-line terminal interface and running Python commands from an interpreter. *Appendix A* covers the basics that you will need to run these examples.

CHAPTER

TWO

INSTALLATION

Gcmstools requires Python and a number of third-party packages. Below is a complete list of packages and minium versions:

- Python >=2.7 (3.x versions not yet supported)
- Pip >=6.0.6 (might be part of new Python releases)
- Setuptools >=11.3.1 (might be part of newer Python releases)
- Numpy >=1.9.1
- Matplotlib >= 1.4.2
- netCDF4 >=1.0.4
- PyTables >=3.1.1
- Scipy >=0.14.0
- Sphinx >=1.2.2 (Optional for documentation.) - numfig is a Sphinx extenstion that is needed to autonumber figures

  references in the docuementation.

Although not required, IPython (v 2.3.1 tested) provides a very useful advanced interactive Python interpreter, and examples in this documentation assume that you are using this environment. See the *IPython* section of *Appendix A* for more details.

2.1 Python

Python and the necessary third-party packages can easily be installed using the all-in-one Anaconda Python distribution. It combines a large number of Python packages for scientific data analysis and a program (`conda`) for managing package updates (in addition to many other advanced features). The Anaconda developers (Continuum Analytics) provide a lot of useful documentation for installing Anaconda and using conda. There are other ways to install Python and it's packages, but for this documentation, it will be assumed that your are using Anaconda.

Note: On Mac/Linux, Python is already part of the operating system. Do not try to install these third-party packages into the builtin Python distribution unless you really know what you are doing. You might overwrite and important file, which can cause problems for your system. Confusion between the system and Anaconda Python installation is a common source of problems for beginners, so make sure that your Anaconda Python is "activated" before running the commands in this document. (See the Anaconda documentation for more information on the activation process.)

Note: On Windows, Anaconda may not install netCDF4. In this case, you can get a prebuilt installer from Christoph Gohlke: be sure to get the Python 2.7 ("cp27") 64-bit ("amd64") build for the most recent version.

Learning the usage of all of these Python packages is far beyond the scope of this document. However, excellent documentation for most of the packages as well as full tutorials are easily discovered.

2.2 gcmstools

To install *gcmstools* from the main repository, there are two options: 1) download the source file and install the package or 2) install using `git` (recommended).

Option 1

Download a zip file of the current state of the repository. (Look for the button shown below (Figure 2.1)at the main repository.) Unzip this package wherever you'd like.

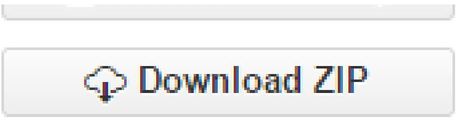

Figure 2.1: Figure 10: The zipfile download button.

From the command line, navigate the newly extracted folder and use `pip` to install the package. In this case, *path-to-gcmstools-folder* is the location of the newly unzipped *gcmstools* folder. Be sure to put a `.` at the end of that pip command.

```
home>$ cd path-to-gcmstools-folder
gcmstools>$ pip install .
```

Option 2 (recommended)

First, install the version-control software Git. *gcmstools* can now be downloaded and installed with one command.

```
home>$ pip install git+https://github.com/rnelsonchem/gcmstools.git
```

The advantage here is that the same command will update your *gcmstools* installation with any any changes that have been made to the main repository.

Uninstall

Uninstallation of *gcmstools* is trivial.

```
home>$ pip uninstall gcmstools
```

CHAPTER

THREE

BASICS OF WORKING WITH GCMS DATA FILES

3.1 Set up the processing environment

In these examples, we will run *gcmstools* form a *terminal IPython* session in a folder “gcms”, which is located in your home directory.

```
home>$ cd gcms

gcms>$ ipython
Python 2.7.9 (default, Oct 10 2014, 15:29:52)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython’s features.
%quickref -> Quick reference.
help      -> Python’s own help system.
object?   -> Details about ’object’, use ’object??’ for extra details.

In :
```

Some example files are provided with the *gcmstools* installation. These files can be moved into the current directory using the `get_sample_data` function.

```
In : from gcmstools.general import get_sample_data

In : get_sample_data()
```

The invocation above copies all of the example files to the current directory. Individual file names can be passed to this function, if you only want a few data files.

```
In : get_sample_data(’datasample1.CDF’)
```

3.2 Note on Conventions

There are potentially many types of GCMS files. However, all importing objects discussed in this section should have identical properties. This is important for later sections of the documentation, because fitting routines, etc. usually do not require a specific file type importer. All of the import objects are constructed with a single string input, which is the name of the file to process. This file name string can also contain path information, if the file is not located in the current directory.

3.3 AIA Files

3.3.1 Export the data

AIA, ANDI, or CDF are all related types of standard GCMS files that are all derived from the Network Common Data Format (netCDF). They may have the file extension "AIA" or "CDF". This file type may not be the default for your instrument, so consult the documentation for your GCMS software to determine how to export your data in these formats.

3.3.2 Read AIA data files

First of all, you will need to import the AIA file reader `AiaFile` from `gcmstools.filetypes` module. To read a file, you can create a new instance of this object with a filename given as a string.

```
In : from gcmstools.filetype import AiaFile

In : data = AiaFile('datasample1.CDF')
```

The variable `data` now contains our processed GCMS data set. You can see its contents using *tab completion* in IPython.

```
In: data.<tab>
data.filename data.intensity data.tic
data.index data.masses data.times
```

Most of these attributes are data that describe our dataset. You can inspect these attributes very easily in IPython by just typing the name at the prompt.

```
In : data.times
Out:
array([0.08786667, ..., 49.8351])

In : data.tic
Out:
array([158521., ..., 0.])
```

This is a short description of these initial attributes:

- *filename*: This is the name of the file that you imported.
- *times*: A Numpy array of the times that each MS was collected.
- *tic*: A Numpy array of the total ion chromatogram intensities.
- *masses*: A Numpy array the masses that cover the data collected by the MS.
- *intensity*: This is the 2D Numpy array of raw MS intensity data. The columns correspond to the masses in the `masses` array and the rows correspond to the times in the `times` array.

The *index* method is used for finding the indices from an array. Its usage is described later.

3.4 Simple plotting

Now that we've opened a GCMS data set. We can easily visualize these data using the plotting package Matplotlib. As an example, let's try plotting the total ion chromatogram. In this case, `data.times` will be our "x-axis" data, and `data.tic` will be our "y-axis" data.

```
In : import matplotlib.pyplot as plt

In : plt.plot(data.times, data.tic)
Out :
[<matplotlib.lines.Line2D at 0x7f34>]

In: plt.show()
```

This should produce a pop-up window with an interactive plot, Figure 3.1. (This should happen fairly quickly. However, sometimes the plot window appears behind the other windows, which makes it seem like things are stuck. Be sure to scroll through your windows to find it.) The buttons at the top of the window give you some interactive control of the plot. See the Matplotlib documentation for more information.

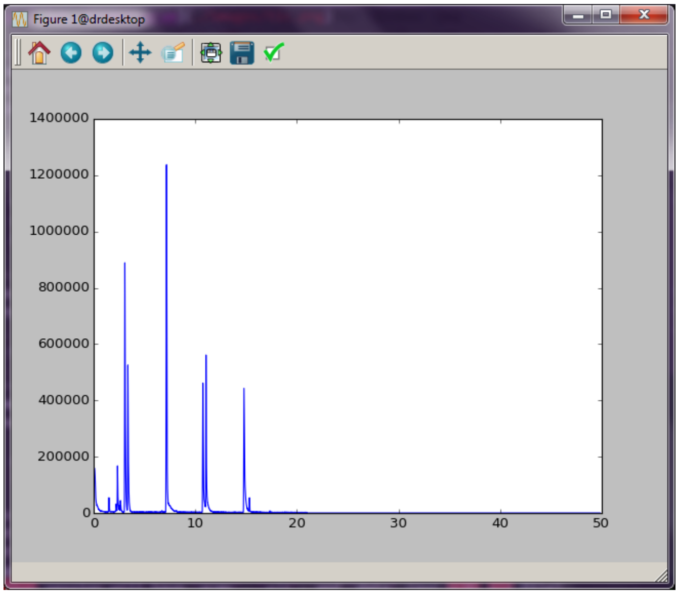

Figure 3.1: Figure 27: Total ion chromatogram.

One drawback here is that you have to type these commands every time you want to see this plot. There is another alternative, though. You can also put all of these commands into a text file and run it with Python directly. Copy the following code into a plain text file called “tic_plot.py”. (See *Working with Text Files* for more information on making Python program files.) In this case, we’re using an AIA file type import, but you can change this for others as you see fit.

```
import matplotlib.pyplot as plt
from gcmstools.filetypes import AiaFile

data = AiaFile('datasample1.CDF')
plt.plot(data.times, data.tic)
plt.show()
```

It is common practice to do all imports at the top of a Python program. That way it is clear exactly what code is being brought into play. Run this new file using the `python` command from the terminal.

```
gcms>$ python tic_plot.py
```

The plot window will now appear, and you will not be able to work in the terminal until you close this window. Alternatively, you can run this program directly from IPython.

```
In : %run tic_plot.py
```

This has the advantage that once the window is closed, you are dropped back into an IPython session that "remembers" all of the variables and imports that you created in your program file. See *Appendix A* for more information here.

3.5 Working with multiple data sets

In the example above, we opened one dataset into a variable called `data`. If you want to manipulate more than one data set, the procedure is the same, except that you will need to use different variable names for your other data sets. (Again, using AiaFile importer as an example, but this is not required.)

```
In : data2 = AiaFile('datasample2.CDF')
```

These two data sets can be plot together on the same figure by doing the following:

```
In : plt.plot(data.times, data.tic)
Out:
[<matplotlib.lines.Line2D at 0x7f34>]

In: plt.plot(data2.times, data2.tic)
Out:
[<matplotlib.lines.Line2D at 0x02e3>]

In: plt.show()
```

The window shown in Figure 3.2 should now appear. (There is a blue and green line here that are a little hard to see in this picture. Zoom in on the plot to see the differences.)

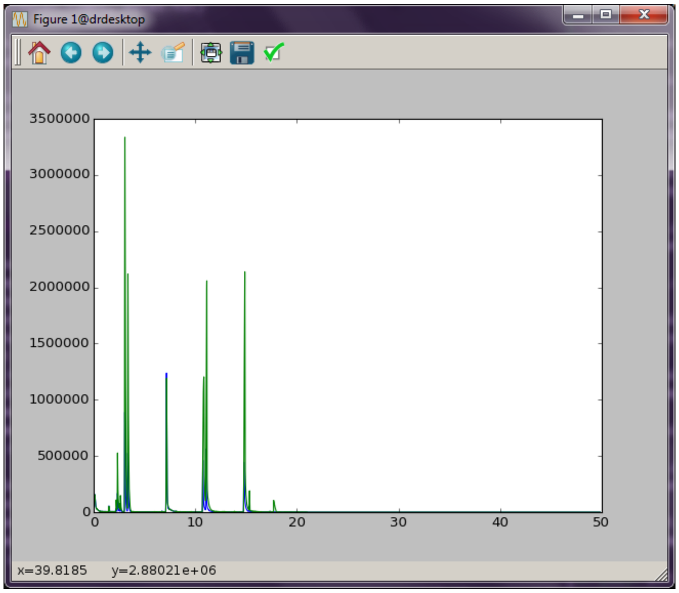

Figure 3.2: Figure 28: Two tic plotted together

CHAPTER

FOUR

NON-NEGATIVE LEAST SQUARES FITTING

4.1 Collecting Reference MS Files

A series of reference spectra are required if you want to do non-negative least squares (NNLS) fitting of your data. There are two example files in this repository to use as reference: “ref_spec.txt” and “ref_spec2.MSL”. The MSL file is a type of text file library that can be exported by programs like the AMDIS or the NIST Mass Spectral Database. The .txt file was hand generated; more information on this file format is provided in the next paragraph. The format of both of these files is very important; however, MSL files are typically autogenerated by external software and may not need manual modification. A common feature of both formats, though, is that comment lines can be included by starting a line with `#`. This can be useful if you want to add some notes or to remove reference spectra without deleting them entirely.

Hand generated reference files must be text files that end with the prefix ”.txt”. Each reference compound must have at the minimum two labels: “NAME” and “NUM PEAK”. “NAME” will be the reference name (probably want this to be concise), and “NUM PEAKS” is followed by (at least) two space-separated columns of MS data. The first column are m/z values, and the second column are the intensity values. Intensities are normalized on import; it is not necessary to do this by hand. Other labels can also be included if you would like to incorporate extra metadata about the reference compound. Each reference compound *must* be separated by a blank line. Below is a small sample of one of these files:

```
NAME:octane
FROM:www.massbank.jp
ID_NUM: JP004695
NUM PEAKS:
  42 14.07 141
  43 99.99 999
  44 2.54 25
  45 4.03 40
  53 1.58 16
  55 19.83 198
  .
  .
  .
```

The online MS repository massBank is a useful place to find these mass and intensity values. The data from that site is already formated correctly for this file type.

4.2 Loading Reference Spectra

In order to import this reference file, we will use the AIAFile object’s `ref_build` function. Below is a repeat of the steps necessary in IPython to set up our environment. (These are unchanged from the previous sections but are repeated for clarity.)

```
In: import matplotlib.pyplot as plt
In: import gcms
In: data = gcms.AIAFile('data/datasample1.CDF')
```

At this point, we are ready to read in our reference file using the function `AIAFile.ref_build`.

CHAPTER

FIVE

AUTOMATED CALIBRATION AND INTEGRATION

5.1 Calibration Data

If you have calibration data for a particular reference compound, you must create a csv file and folder that have the same base name as the reference MS file from above. Again, an examples are provided in this repository called refcpd.csv and the folder refcpd. All of your calibration AIA files for this compound need to be stored in the newly created folder. In order for these new data files to be processed, the refcpd.csv file must be appropriately modified.

The csv file is a simple comma-separated text file, but again the structure is important. The first row in this file is critical. At the end, there are two values that define the starting and stopping time points for integration. Change these values based on the time range that you've determined from the TIC of a calibration run. The rest of the rows are data file information. The first column is the name of a calibration data file, and the second column needs to be the concentration of the reference compound associated with that run. You don't have to add all of the calibration files here, but if they are not in this list, they won't be processed. Alternatively, any line that starts with a '#' is a comment, and will be ignored. In this way, you can comment out samples, and add some notes as to why that sample was not used or whatever.

5.2 Run Calibrations

Once you've updated the calibration information from above. You can run the program 'calibration.py'. This runs through all of the reference spectra defined in the 'reference_files.txt' file. If a '.csv' file exists for a particular reference file, then a calibration will be performed.

All of the calibration data files listed in the csv file will be processed and a calibration curve generated. For each calibration sample, a plot of the reference-extracted data will be generated in the calibration folder (refcpd_fits.png). In addition, a calibration curve plot is also generated (refcpd_cal_curve.png'), which plots the integrated intensities and calibrated intensities vs the concentrations. In addition, the calibration information is printed on the graph for quick visual inspection. There is no need to write down this calibration information.

This program has some important command line arguments that will change the programs defaults. The first argument, '–nobkg', is a simple flag for background fitting. By default, the fitting routine will select a MS slice from the data set and use that as a background in the non-negative least squares fitting. This procedure can change the integrated values. If you use this flag, then a background MS will not be used in the fitting. Using a background slice in the fitting may or may not give good results. It might be a good idea to look at your data with and without the background subtraction to see which is better.

The second command line argument is '–bkg_time'. By default, the fitting program uses the first MS slice as a background for fitting. However, if there is another time that looks like it might make a better background for subtraction, then you can put that number here.

Here's a couple of example usages of this script:

```
# This will run the calibration program with all defaults
$ python calibration.py
# This shuts off the background subtraction
$ python calibration.py --nobkg
# This sets an alternate time for the background subtraction
# In this case, the time is set to 0.12 minutes
$ python calibration.py --bkg_time 0.12
```

Another file is also generated during this process: cal.h5. This is a HDF5 file that contains all of the calibration information for each standard. Do not delete this file; it is essential for the next step. This is a very simple file, and there are many tools for looking at the internals of an HDF5 file. For example, ViTables is recommended. The background information, such as whether a background was used and the time point to use as a background spectrum, are stored as user attributes of the calibration table.

CHAPTER

SIX

PROCESS SAMPLE DATA

Put all of your data files in a folder that must be called ‘data’. Once you’ve done this, run the program ‘data.py’ to process every AIA data file in that folder using the calibration information that was determined from the steps above.

This program opens the AIA file for the sample and performs non-negative least squares analysis of the full data set using the reference spectra that are listed in the ‘reference_files.txt’ file. Using the calibration information that was determined above, it finds the concentrations of those components in the sample data. For every reference compound that has associated calibration information, a plot is generated that overlays the TIC (gray) and extracted reference fit (blue). The title of the plot provides the calibrated concentration information. Visual inspection of these files is recommended.

This file also accepts the same command line arguments as ‘calibration.py’ from the section above. You will be warned if you try to analyze your data with different background information than the calibration samples. This may not impact your data much, but it is good to know if you are doing something different.

This file also generates another HDF5 file called ‘data.h5’, which contains the integration and concentration information for every component. This information is identical to what is printed on the extraction plots above. However, this tabular form of the data is a bit more convenient for comparing many data sets. See the Calibration section for a recommended HDF5 file viewer.

CHAPTER SEVEN

APPENDIX A: RUNNING CODE SAMPLES

7.1 Using the Command Line

Running the code samples in this documentation requires a rudimentary knowledge of using command-line terminal (command prompt on Windows). The terminal can seem very “texty” and confusing at first; however, with a little practice it gets to be fairly intuitive and *efficient*. There are many tutorials online. See The Command Line Crash Course, for example.

A few basics are covered there, though, for reference. When you start a terminal, you will be presented with a little window where you can type commands. In this documentation, the terminal command prompt will be denoted as `home>$` in this document. Where *home* is simply the current folder where the commands will be executed, and *>$* is just a separator. These things do not need to be typed when entering commands. Some commands generate output. The output will occur after the command prompt, but will not be preceded by a command prompt symbol.

A couple of useful command line options are “change directory” (`cd`) and “present working directory” (`pwd`). The most important thing you’ll want to be able to do is move directories (i.e. folders). When you open the terminal, you will usually start in your *home* directory. This will probably be “/Users/username” on Mac, “/home/username” on Linux, or “C:Usersusername” on windows. To move to a different directory, use the `cd` command: to find out the location of the current folder use `pwd`. Here’s an example.

```
home$> pwd
/home/username/

home$> cd folder1

folder1$> pwd
/home/username/folder1
```

The second command here moved active directory to the folder “folder1”. There are a few special directory shortcuts:

- ~ : This refers to the home directory.
- `..` : (Double period) This refers to the base directory of the current directory.
- `.` : (Single period) This refers to the current directory.
- `\` or `/` : Separators to combine directory names. The first works on Linux/Mac, the second is required on Windows.

Here’s these shortcuts in action.

```
home>$ cd folder1/folder2

folder2>$ pwd
/home/username/folder1/folder2
```

```
folder2>$ cd .

folder2>$ pwd
/home/username/folder1/folder2

folder2>$ cd ../..

home>$ pwd
/home/username

home>$ cd folder1/folder2

folder2>$ cd ~

home>$ pwd
/home/username

home>$ cd folder1/folder2

folder2>$ cd ~/folder3

folder3>$ pwd
/home/username/folder3
```

The other important command is “list” (`ls` or `dir` on Windows). This lists the contents of the current directory.

```
folder3>$ ls
file1 file2 folder4
```

7.2 IPython

7.2.1 Start IPython

One of the strengths of Python as a data analysis language is its interactive interpreter. This mode is accessed from a terminal by typing `python`.

```
home>$ python
Python 3.4.1 (default, Oct 10 2014, 15:29:52)
[GCC 4.7.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Any command that is typed at the >>> prompt is treated as Python code, and executed appropriately. That means you can interactively write and explore code in this manner.

```
>>> 2 + 2
4
>>> print('Hello World')
Hello World
```

The default Python interpreter is very limited, which is why IPython was developed. IPython is an advanced Python interpreter, which gives you things like autocompletion and introspection, just to name two. In addition, this project has grown substantially, and in addition to a terminal based interpreter, there is also a GUI version and a very cool web-based Notebook as well. In this documentation, we are only going to use the terminal version of IPython. To learn more about the other features, consult the IPython documentation.

IPython is started from the terminal using the `ipython` command:

```
home>$ ipython
Python 3.4.1 (default, Oct 10 2014, 15:29:52)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython’s features.
%quickref -> Quick reference.
help      -> Python’s own help system.
object?   -> Details about ’object’, use ’object??’ for extra details.

In [1]: 2 + 2
Out[1]: 4

In [2]: print(’Hello World’)
Hello World
```

The `In [#]:` prompt now takes the place of the `>>>` prompt in the regular Python interpreter. In addition, certain types of output are preceded by the `Out[#]:` prompt. The numbers in brackets help you to determine the order that commands are processed. For this documentation, though, the numbers will be striped for clarity: e.g. `In :` and `Out:` will denote commands run and output, respectively, in an IPython interpreter.

7.2.2 Autocompletion and Introscpection

Two very nice aspect of the IPython interpreter are autocompletion and object introspection. Both of these will make use of the Tab key on your keyboard; in code snippets, this key will be denoted as `<tab>`, which means you should press the Tab key rather than typing it out. To see these two operations in action, we can first create a new string object.

```
In : my_string = ’Hello World’

In : print(my_string)
Hello World
```

To determine the methods available to a string object, we can use IPythons object introspection.

```
In : my_string.<tab>
my_string.capitalize    my_string.isidentifier  my_string.rindex
my_string.casefold      my_string.islower       my_string.rjust
my_string.center        my_string.isnumeric     my_string.rpartition
my_string.count         my_string.isprintable   my_string.rsplit
my_string.encode        my_string.isspace       my_string.rstrip
my_string.endswith      my_string.istitle       my_string.split
my_string.expandtabs    my_string.isupper       my_string.splitlines
my_string.find          my_string.join          my_string.startswith
my_string.format        my_string.ljust         my_string.strip
my_string.format_map    my_string.lower         my_string.swapcase
my_string.index         my_string.lstrip        my_string.title
my_string.isalnum       my_string.maketrans     my_string.translate
my_string.isalpha       my_string.partition     my_string.upper
my_string.isdecimal     my_string.replace       my_string.zfill
my_string.isdigit       my_string.rfind
```

As you can see, there are many, many things that you can do with this string object. IPython can also use the Tab key to autocomplete long names for variables, path strings, etc. Here’s an example:

```
In : my_string.is<tab>
my_string.isalnum       my_string.isidentifier  my_string.isspace
my_string.isalpha       my_string.islower       my_string.istitle
my_string.isdecimal     my_string.isnumeric     my_string.isupper
my_string.isdigit       my_string.isprintable

In : my_string.isi<tab>
```

Notice that when you type tab here IPython automatically expands this to `my_string.isidentifier`. This works for path strings as well.

The take home message here is: *use the Tab key a lot!* It will make you much more productive.

Note: It should be pointed out that tab completion also works on the regular command line terminal interface as well.

7.2.3 Magic Commands

IPython has a number of special commands that make the IPython interpreter behave much like a command-line terminal. These commands, called Magic Commands, are preceded by `%` or `%%`. The magic command documentation covers most of these commands, but a few that are useful to the examples in this document are discussed here.

The magics `%cd`, `%pwd`, and `%ls` serve the exact same purpose as in the terminal. A very useful magic, though, is `%run`, which can be used to execute a Python program file from inside the IPython session. This is best explained by example. Create a new folder called `folder1` in your home directory. Create the file `test.py` in `folder1` and paste the following code into that file. (See *Working with Text Files* for some information on text files and Python programs.)

```
var1 = 7
var2 = "Hello World"
var3 = var1*var2
```

Now let's start up IPython and run this new program.

```
home>$ ipython
Python 3.4.1 (default, Oct 10 2014, 15:29:52)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In : %pwd
/home/username

In : %cd folder1
/home/username/folder1

In : %ls
test.py

In : %run test.py

In :
```

At this point, it seems like nothing has happened; however, the variable that we defined in our file "test.py" are now contained in our IPython session. Assuming that the following IPython code is the same session as above.

```
In : var1
Out: 7

In : var3
Out: Hello WorldHello WorldHello WorldHello WorldHello WorldHello
WorldHello WorldHello World
```

As you can see, this is a very powerful way to save your work for later or run code that is fairly repetitive.

7.3 Working with Text Files

There are many instances where you will need to work with plain text files, including when writing Python programs. Plain text files are *not* word processing documents (e.g. MS Word), so you will want to use a dedicated text editor. Another source of problem for beginners is that leading white space in Python programs is important. For these reasons, a dedicated Python text editor can be very useful for beginners. Anaconda is bundled with Spyder, which has a builtin text editor. The Anaconda FAQ has information on running Spyder on your system. Spyder is actually a full development environment, so it can be very intimidating for beginners. Don't worry! The far left panel is the text editor, and you can use that without knowing what any of the other panels are doing. Some internet searches will reveal other text editors if you'd prefer something smaller. (Do *not* use MS Notepad.)

The ".py" suffix for Python programs is usually not important. However, on Windows at least, this file extension will make it possible to double-click to run the program, which is a feature that some people prefer.