

# SAS: A Mini-Manual for ECO 351

## by Andrew C. Brod

### **1. Introduction**

This document discusses the basics of using SAS to do problems and prepare for the exams in ECO 351. I decided to produce this little guide instead of having you purchase a supplementary text. However, those of you who are considering further work in applied economics might benefit from such a text, and here's a suggestion. The SAS Institute produces a nice volume called *SAS System for Regression* (3rd ed):

<http://www.sas.com/apps/pubscat/bookdetails.jsp?pc=57313>

The book is also available at Amazon.com and elsewhere.

### **2. Starting and Operating a SAS Session**

If you're on campus, look for SAS 9.1.2 in the Application Explorer, clicking first on Academic Software and then on Data Analysis Tools. It often takes a little while to load.

The SAS environment comprises a series of windows:

**Log:** SAS prints error messages and warnings here;

**Output:** SAS prints results of your programs here;

**Editor:** use this to write and edit programs;

**Explorer:** use this to browse folders and drives on your computer and the Novell network;

**Results:** use this to manage results from different programs, and to maneuver around the output window.

You can use the buttons on the status bar at the bottom if one of these windows is hidden. Of course, SAS has the usual array of Microsoft-based commands and menus to manage files, print output, clear a window, obtain help, and so on.

To create a SAS program, just start typing it in the Editor window (SAS has other editing options, including an enhanced editor and what it calls the SAS Notepad). You can submit it via the Run menu, and save it when you're done. SAS programs use the **.sas** extension. You can use the Editor to create or edit a data file as well. It turns out that it's quite easy to copy data from the files on the class website, paste them into the SAS editor, and then save the resulting file as you wish.

### **3. Data and Proc Steps**

The point-and-click interface of SAS 9 is great for file management. But SAS programming and analysis uses a more old-fashioned approach. First you obtain a data set and get it into the appropriate format. Then you write a SAS program, a sequence of commands strung together in a logical progression. The program accesses the data file, performs a series of statistical operations on the data, and then generates usable output. The order of the commands depends on the nature of the task to be performed.

As an example, consider this simple SAS program:

```
data firsttry;  
  input x y @@;  
  cards;  
    2 5 4 7 6 4 8 8 10 12 12 13  
proc print;  
  var x y;  
proc means; var x y;  
run;
```

In fact, you can copy and paste this very program into the Editor window and run it to see what happens. The basic structure of this program is:

```
data step  
proc step  
proc step  
run
```

Regardless of the analysis to be performed, there has to be data on which to perform it. Therefore, every SAS-language program must begin with a **data** step, which reads in the data and, if appropriate, transforms it (e.g. by taking the logarithm of a variable or by subtracting one variable from another—more on this later). There may be more than one **data** step in a SAS program, but the program must begin with a **data** step (the only exceptions are formatting commands like **options**). A **data** step is made up of a **data** command, plus any commands that follow it up until a **proc** command or another **data** command. In the above program, the single **data** step is made up of two commands, **input** and **cards** (more on these later), plus a line of data.

Once the data are read in and are in the desired form, it's time to analyze them. For this one employs one or more **proc** steps. A **proc** step is made up of a **proc** command, which just gives the name of the procedure being run, plus any commands that follow it up until another **proc** or **data** command. The two procedures in the above program tell SAS to perform two simple tasks: print out the data that were just read in; and calculate some sample statistics. To accomplish this, the two **procs**, **print** and **means**, appear after the **data** step. Each of these **procs** includes a **var** instruction to tell SAS which variables to analyze.

When the desired tasks are completed, there's nothing else to tell SAS except that the program is over. The **run** command, which must appear at the end of the program, does this.

In my description of the simple SAS program, I've talked about the sequence of tasks (read in the data, print them out, calculate sample statistics) as though I were doing them in "real time." But that's not really how SAS works. You must write the program *as if* you were doing the tasks in real time, but SAS doesn't actually execute the commands until you run the program. Writing a SAS program is like writing an essay. To write well, you have to put yourself in the place of the reader and ask things like: "Would this sentence make sense if I didn't already know what I'm trying to say?" When writing a SAS program, you have to put yourself in the place of that unimaginative, but extremely obedient, entity called SAS. How will SAS interpret the commands? If you put the **proc print** before the initial **data** step, then you would in effect be telling SAS to print out the data without first telling it what or where the data are! And that wouldn't make sense, would it?

Note that each command in a SAS program *must* end with a semi-colon (except the line containing the data, which isn't really a command). The semi-colon is how SAS knows the command is over, and it's what allows you to put two commands on the same line (as I did above in my **proc means**). Most commands that you will use will be part of either a **data** step or a **proc** step. One exception we've seen already is the **run** command.

#### **4. Entering and Reading in Data**

*Entering data directly in your SAS program:* The above sample program employs a data set which is small enough to be typed in and included with the program. The **input** command in that program tells SAS that two variables, **x** and **y**, are to be read in, and the **cards** command tells SAS that the data will be included with the program instead of read from an external data file. The double @ tells SAS that once the first **x** and **y** are read in, it should continue to read that line as long as there are data to read. If you looked in the Output window after running the above program, the **proc print** output would look something like:

<b>Obs</b>	<b>x</b>	<b>y</b>
<b>1</b>	<b>2</b>	<b>5</b>
<b>2</b>	<b>4</b>	<b>7</b>
<b>3</b>	<b>6</b>	<b>4</b>
<b>4</b>	<b>8</b>	<b>8</b>
<b>5</b>	<b>10</b>	<b>12</b>
<b>6</b>	<b>12</b>	<b>13</b>

However, if you omitted the double @ from the program, you'd get only the following in the Output window:

Obs	x	y
1	2	5

...and nothing more! SAS would stop reading after the first (x,y) combination. In order to read the data in without the double @, you'd have to use the following **data** step in which the data are typed in columns instead of rows:

```
data firsttry;
  input x y;
  cards;
  2 5
  4 7
  6 4
  8 8
  10 12
  12 13
```

This would yield the correct output.

*Reading data in from an existing file:* The homeworks will often require you to use data sets I have set up, which you can read with the **infile** command. Suppose the data in the above example are contained in a file called **stat.dat**. Then the **data** step of the above program could be:

```
data firsttry;
  infile 'stat.dat';
  input x y;
```

Note that **infile** takes the place of **cards**, and it goes *before* **input**, not after it. In addition, the name of the data file must be in single quotes. Whether you need the double @ or not in this INPUT command depends on how the data are arranged in the data file: in a row or in columns.

The **infile** command is a bit tricky to work with, because you have to give SAS the precise location of the data file. The above **infile** command assumes that **stat.dat** is in the same directory as you're currently working in. But data files could be anywhere. If **stat.dat** is on a disk in your a: drive, then you'd use the following **infile** command:

```
infile 'a:stat.dat';
```

The main point is that SAS allows you to point and click to your heart's content *except* when you're writing a SAS-language program, where you have to get the location of the data file *exactly* correct. It's a bit of a pain, but it's the way it is.

It's always a good idea to take a look at the data file before you write your **infile** command. In addition to seeing how the data are arranged (in rows or in columns), you may find a line or two of labels at the top or the bottom of the data file which you don't want SAS to read as data. The easiest way to ensure this is to edit the file and then use the edited version in your SAS program.

Another way is to use the **firstobs** and **obs** options on the **infile** command. If **stat.dat** has one line of labels at the top, you tell SAS to start reading on its 2nd line by using:

```
infile 'a:stat.dat' firstobs=2;
```

where I've assumed that **stat.dat** is on your a: drive. In general, using **firstobs=n** tells SAS to start on line #n. Leaving it off altogether is the same as using **firstobs=1**. Setting **obs=m** tells SAS to stop reading after line #m, and leaving the **obs** option off altogether is the same as telling SAS to read to the end of the file. So if **stat.dat** has 3 lines of labels, then 20 lines of data, then 8 lines of data definitions, and you don't want to edit the data file, you can tell SAS to read only the data by using:

```
infile 'a:stat.dat' firstobs=4 obs=23;
```

*Column numbers in the INPUT command:* Suppose the variables **x**, **y**, and **z** are arranged in columns in an external data file, but you only want to analyze **x** and **z**. One option would be to simply read in all three variables after the **infile** command:

```
input x y z;
```

and then just ignore **y**. Alternatively, if you knew that **x** was recorded in columns 1-6 in the data set, with **y** in columns 7-14 and **z** in columns 15-16, you could do the following:

```
input x 1-6 z 15-16;
```

If the data file contains a large number of variables, this feature can be quite handy.

*Character variables:* Numeric variables are variables that take on numeric values like 0, 1, 78, and -3.46319. Such variables can be added, subtracted, multiplied, and otherwise manipulated mathematically. Character variables have as their values strings of characters, usually (but not necessarily) including letters. Character variables are non-numeric and hence cannot be manipulated mathematically. As an example, consider the following **data** step:

```
data packages;
```

```
input sales design $ @@;  
cards;  
4.58 new 3.97 old 5.19 new 4.88 old  
5.27 new 4.96 old 5.84 new 5.13 old
```

The **\$** code in the **input** statement tells SAS that the variable **design** is a character variable; **sales**, which has no such code, is designated as a numeric variable. From the inputted data, we see that **design** takes on values ‘new’ and ‘old’ (note the single quotation marks around the character strings). We can, for example, add 4.58 and 3.97, the first two values of **sales**, but it’s meaningless to try to add ‘new’ and ‘old’, the first two values of **design**.

## **5. Printing and Displaying Data**

*Proc print:* Once you read data in, it’s often a good idea to print them out right away to make sure you read them in correctly. Or you might want to print out the results of a statistical analysis. In either case, you’d use **proc print**, which has the following structure:

```
proc print data=SAS-data-set;  
var variable-list;  
by variable-list;  
title ‘title’;
```

You need to use the **data=** option only when the data set you’re printing was *not* the one most recently created. In the sample program from Section 4 of this document, there is no need to specify **data=firsttry** because **firsttry** is the only data set used in that program. But there are times when greater specificity is desirable.

The variable list is also optional, depending on what you’re trying to do. In the sample program, the data set **firsttry** contained the variables **x** and **y**. If you omit the **var** command in **proc print**, then SAS will print out *all* the variables in the data set. In my sample program, I wanted to print out both **x** and **y** (which is all the variables in that small data set), so I could have simply left the **var** off. If I’d wanted to print out only the variable **x**, I would have had to use

```
var x;
```

The **by** command, which can be used in any **proc**, performs a separate analysis for each value of the variables in the **by** command’s variable-list (in this case, the “analysis” is merely printing). If variable **z** is a “dummy” variable that equals either zero or one, then

```
proc print; var x; by z;
```

creates two separate print-outs, one for those observations for which  $z=0$  and one for  $z=1$ . The only hitch here is that the data must be sorted for this feature to work, that is, all observations for which  $z=0$  should be together and they should all come before any observations with  $z=1$ . To do the sorting, place the following commands before the **proc print** command:

```
proc sort; by z;
```

The **title** command, which also can be used in any **proc**, allows you to place a title on your output. You could do something like one of the following:

```
title 'Regression Output for Assignment #2';  
title 'Andrew Brod is a wonderful human being';
```

Whatever you place inside the single quotes is printed at the top of each page of your output. Once you specify a title, any output your program generates from that point on will have that title. To discontinue a title in the middle of a program, simply specify an “empty” title:

```
title ' ';
```

*Proc gplot:* A numerical print-out can be quite informative, but sometimes it helps to graph, or plot, the data. SAS has two main plotting procedures, **plot** and **gplot**. **Proc plot** was more useful in the past, when graphics were harder to manage, so let's use **proc gplot** for this course. It's relatively easy to insert **gplot** diagrams into, say, a Word document. Here's the basic structure:

```
proc gplot data=SAS-data-set;  
plot plot-list / options;
```

Once again, you don't have to specify the SAS data set if you're analyzing the most recently created data set. The “plot-list” in the **plot** command tells SAS what variables to plot. For example, to create a scatter diagram in which variable **y** is plotted against variable **x**, you would use:

```
plot y*x;
```

This plot measures **y** along the vertical axis and **x** along the horizontal axis. To create a time plot of variable **y**, you would have to have already defined a time variable (creating such variables is covered below), which might be called **time**. Then you would use:

```
plot y*time;
```

There are many options governing the plot's appearance, and SAS's on-line documentation goes into them all in excruciating detail.

## **6. Important Statistical PROCs**

*Proc means*: The simplest statistical procedures are **means** and **univariate**, which calculate univariate statistics. The structure of **proc means** is:

```
proc means data=SAS-data-set statistic-list;  
var variable-list;
```

As with all other **procs**, you need not specify the data set as long as you want to analyze the one most recently created. If you omit the **var** command, SAS will perform the analysis on every variable in the data set. If no statistic-list is included in the **proc means** statement, then the following default statistics are automatically calculated for each variable in the variable-list (assuming that “y” is the observed variable):

**n**: the number of observations (n)  
**mean**: the sample mean ( $\bar{y}$ )  
**std**: the sample standard deviation (s)  
**min**: the minimum value of y observed  
**max**: the maximum value of y observed

But other statistics are available, including:

**sum**: the sum of all values in the sample  
**var**: the sample variance ( $s^2$ )  
**stderr**: the standard error of the mean ( $s/\sqrt{n}$ )  
**uss**: the “uncorrected” sum of squares  
**css**: the “corrected” sum of squares  
**cv**: the coefficient of variation ( $s/\bar{y}$ )  
**t**: the t-statistic for the test of  $H_0: \mu = 0$  vs.  $H_1: \mu \neq 0$   
**pvt**: the p-value for the above test

SAS calculates its uncorrected and corrected sums of squares as follows:

$$USS = \sum_{i=1}^n y_i^2 \quad CSS = \sum_{i=1}^n (y_i - \bar{y})^2$$

Therefore, what SAS means by “correcting” the sum of squared observations is first subtracting off the sample mean before squaring and summing.

If you want any statistics other than those in the default list, you need to include them in the **proc means** statement. But if you do so, you need to specify any and all statistics you want reported. For example,

**proc means cv;**

does *not* tell SAS to report the default statistics (**n, mean, std, min, max**) *plus* the coefficient of variation; it tells SAS to report *only cv*!

Finally, if  $\mu$  is the unknown population mean of  $y$  (i.e.  $E(y) = \mu$ ), then the t-statistic reported by **proc means** is the one for the two-tailed test that  $\mu$  is zero. If you want to test whether  $\mu$  takes on any value other than zero, you need to transform the original data accordingly (data transformations are discussed below). The p-value **prt** is the probability of observing, given that  $H_0$  is true, a t-value which is bigger in absolute value than the reported value **t**. In symbols, **prt** =  $\Pr\{|t| > T\}$ . The p-value for the one-tailed test of  $H_0: \mu = 0$  vs.  $H_1: \mu > 0$  (or of  $H_0: \mu = 0$  vs.  $H_1: \mu < 0$ ) is one-half of **prt**.

*Proc univariate*: This procedure reports the same statistics as **proc means**, plus a few more, but it does so a bit less flexibly. The syntax of **univariate** is:

```
proc univariate data=SAS-data-set options;  
var variable-list;
```

Therefore, the commands

```
proc univariate; var x y;
```

calculate univariate statistics for variables **x** and **y**. The only option we're likely to use is **normal**. The commands

```
proc univariate normal; var x;
```

tell SAS, in addition to calculating the usual statistics, to perform a test of whether the observed values of **x** could have come from a normal distribution.

As discussed above, any **proc** can include a **by** command. If **z** is a variable taking on two values, 12 and 24, then

```
proc univariate; var x; by z;
```

calculates univariate statistics for two separate subsamples, one for those observations for which  $z=12$  and one for  $z=24$ .

*Proc ttest*: This procedure tests the equality of two population means ( $H_0: \mu_1 = \mu_2$  vs.  $H_1: \mu_1 \neq \mu_2$ ) or two population variances ( $H_0: \sigma_1^2 = \sigma_2^2$  vs.  $H_1: \sigma_1^2 \neq \sigma_2^2$ ). Its basic structure is:

```
proc ttest data=SAS-data-set;  
class variable;
```

**var** variable-list;

For each variable in the **var** statement's variable-list, **proc ttest** compares sample means and variances between two groups, where the groups are determined by the value of the grouping variable named in the **class** statement (which *must* be included in **proc ttest**). The **class** variable must take on exactly two values and it may be a character-valued variable. For example, if **z** takes on values 'up' and 'down', then

**proc ttest; class z; var y;**

tells SAS, among other things, to compare the mean of **y** when **z**='up' to the mean of **y** when **z**='down'.

*Proc freq*: This procedure produces 1-way and 2-way tabulations of discrete data and performs some tests of association. Its basic structure is:

**proc freq data=SAS-data-set;**  
**tables** table-list / options;

Data are called "discrete" when all observed values fall into a small number of categories. If variable **y** is the weight of an adult, then **y** is *not* discrete, it's continuous. A wide range of observed weights is possible, and between any two possible observed weights, say 155 and 156 lbs., there are many other possible observations, such as 155.79 lbs. In contrast, if variable **x** is the number of daily newspapers in a city and the only observed values are 0, 1, and 2, then **x** is discrete.

**Proc freq** counts up the number of observations in each category. If you use, say,

**tables x;**

then SAS will produce a 1-way tabulation of the sample distribution of **x**, a sort of numerical version a bar chart. However, 1-way tabulations are the default in **proc freq**, so omitting the **tables** command does the same thing. In contrast, using

**tables x\*z;**

tells SAS to produce a 2-way cross-tabulation of the sample distributions of **x** and **z**. Suppose that **x** takes on 3 possible values, 0,1,2, and that **z** takes on values 0 or 1. Then SAS will produce a 3×2 contingency table. If you use

**tables x\*z / chisq;**

then SAS will perform a  $\chi^2$  test of independence between **x** and **z**.

*Other procedures:* Procedures that we'll need in order to run and analyze regressions, most notably **proc reg**, will be covered in classes and labs. This document is designed to get you started with SAS, not to cover all you'll need to know for the class.

## **7. Manipulating Data**

The data you read in from a file may require further processing before you can do a meaningful statistical analysis. Any transformations of the data must take place in a DATA step, either the initial one of your program where you read the data in, or another DATA step later in the program.

*Assignment statements:* The simplest type of data transformation is an “assignment statement,” a SAS command that creates a new variable from existing ones. For example, if you have data on miles travelled (**miles**) and the number of hours it took to drive those miles (**hours**), you can calculate a new variable, miles per hour (**mph**):

```
mph = miles/hours;
```

where the slash indicates division. This command *assigns* **mph** to be **miles** divided by **hours**. Here are some sample assignment statements, assuming that **x** and **y** are existing variables:

```
z = y*x;          diff = x - y;          ycube = y**3;  
yroot = sqrt(y);  logy = log(y);         expy = exp(y);       frog = abs(y);
```

where an asterisk (\*) indicates multiplication and a double asterisk (\*\*) indicates exponentiation. The above commands would create **z** as the product of **y** and **x**, **diff** as **x** minus **y**, **ycube** as **y** cubed, **yroot** as the square root of **y**, **logy** as the natural logarithm of **y**, **expy** as the number e (2.7182818) raised to the **y** power, and **frog** as the absolute value of **y**. Why “**frog**”? Why not? You can name your variables anything. However, **sqrt**, **log**, **exp**, and **abs** are specific names for SAS-language functions.

You can use an assignment statement to create a time-trend variable:

```
time = _n_;
```

The variable **\_n\_** is a SAS-defined variable that records the observation number. If your sample has 25 values, then **\_n\_=1** for the first observation, **\_n\_=2** for the second, and so on until **\_n\_=25** for the last observation. But **\_n\_** can only appear in a **data** step; you can never use it in a **proc**, which is why it might be useful to create a variable like **time**. Of course, you could call your time-trend variable anything, including **trend**, **obsnum**, or **bush2004**.

If an assignment statement is located in the same **data** step that reads in the original variables, then its placement is important. Consider the first assignment statement of this

section. If **miles** and **hours** are read in from an external file, then the **data** step should look something like:

```
data speed;  
  infile 'filename';  
  input miles hours;  
  mph = miles/hours;
```

which seems like the logical order for these commands. But if **miles** and **hours** are read in via **cards**, then the assignment statement must go before **cards** along with the **input** command, as in:

```
data speed;  
  input miles hours;  
  mph = miles/hours;  
  cards;  
  <numbers>
```

*Subsetting 'if' statements:* You may find that you want to analyze a subset of a larger data set. Subsetting **if** statements allow you to whittle the large data set down to a more manageable size and work only with selected observations. Suppose you have ratings figures from a sample of TV stations, which you place in variable **view**. But some of the stations are strictly cable stations and some are broadcast-and-cable stations, and suppose you want to analyze only the broadcast stations. If you have a “dummy” variable **cable** that equals 1 if the station is only on cable and 0 if it's broadcasted as well, then you could use the following **data** step:

```
data viewers;  
  infile 'ratings.dat';  
  input cable view;  
  if cable=0;
```

These commands read in variables **cable** and **view** from the data file **ratings.dat** but include in the data set **viewers** only those observations for which **cable=0**, i.e. only the broadcast stations. Subsequent **procs** using the data set **viewers** will be able to analyze the ratings figures only for broadcast stations. Any valid mathematical expression can go in the **if** statement. For example, you could accomplish the same subsetting as the above (i.e. analyzing only broadcast stations) by using the following alternative **if** statement:

```
if cable<1;
```

If **cable** happens to be a character variable with values 'cable' and 'broadcast', then the appropriate **data** step would be:

```
data viewers;
```

```
infile 'ratings.dat';  
input cable $ view;  
if cable='broad';
```

Note the single quotes around the *values* of **cable**.

*If-then-else commands:* You can also use **if** statements to define new variables depending on whether a given criterion is satisfied. For example,

```
if y>10 then z=y;  
else z=0;
```

These commands take an existing variable **y** and define a new variable **z** to be equal to **y** when **y** is greater than 10 but to be equal to zero when **y** is less than or equal to 10. To illustrate, let's alter the program in Section 3 of this document and insert these two commands:

```
data firsttry2;  
input x y @@;  
if y>10 then z=y; else z=0;  
cards;  
2 5 4 7 6 4 8 8 10 12 12 13  
proc print;  
var y z;  
run;
```

This yields the following:

<b>Obs</b>	<b>y</b>	<b>z</b>
<b>1</b>	<b>5</b>	<b>0</b>
<b>2</b>	<b>7</b>	<b>0</b>
<b>3</b>	<b>4</b>	<b>0</b>
<b>4</b>	<b>8</b>	<b>0</b>
<b>5</b>	<b>12</b>	<b>12</b>
<b>6</b>	<b>13</b>	<b>13</b>