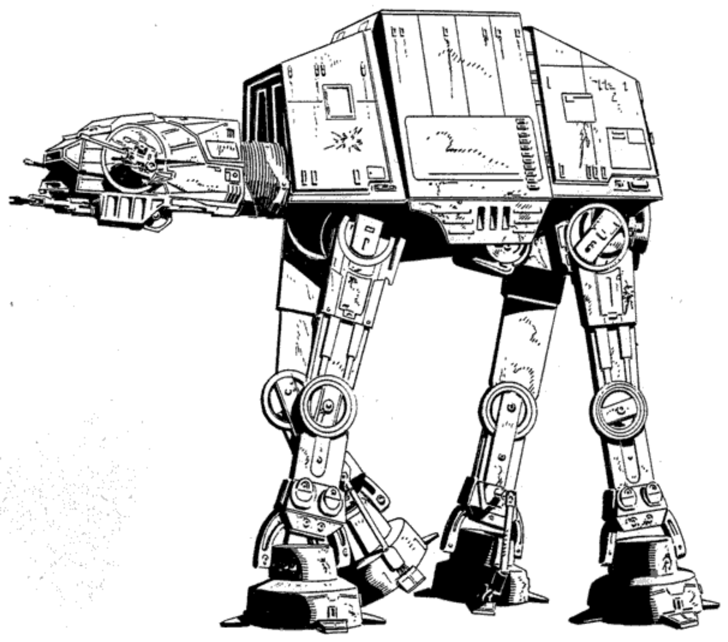
AT-ATS

**Statistical Analysis &
Homework on the Cheap**

Release 0:11:284 Alpha

22 April, 2008



Overview and Frequently Asked Questions

What is AT-ATS?

AT-ATS is a short program written to allow beginning econometrics students to perform basic statistical analysis as required in their homework. It designed to work and look like STATA, but contains none of the same code.

I've just discovered that one can also use it for quick-and-dirty Monte Carlo simulations. I may add some more features to make this even more useful.

Why the stupid name?

It's STATA spelled backwards. Not that that explains the stupidity.

How is AT-ATS like STATA?

AT-ATS tries to use the same syntax and commands as STATA, though it deviates on occasion when STATA's syntax is just ridiculous and/or inconsistent.

How is AT-ATS unlike STATA?

STATA is a big program. It does lots and lots of things. It can do every statistical test you have ever heard of, and many that you have not heard of. AT-ATS, on the other hand, implements only a small subset of STATA's capabilities. Specifically, only the statistical functions that a GSPP student would see in PP240A and PP240B are implemented. In essence, you can do your homework. You can probably do a bit more, but not much more.

Why did you create AT-ATS?

I've never been too impressed with STATA. Probably my biggest gripe is the $100 to buy the software, even for a student. This is especially annoying for two classes of reasons. First, the software is lame. It's text-based with a slapped-on graphical interface. It has inconsistent command syntax and sports an obtuse and proprietary programming language. The second problem is that, as a statistics package, it implements a pile of algorithms that are widely known and published in introductory textbooks everywhere.

In all honesty, however, it was also a fun way to learn stats.

So, what functions are implemented in AT-ATS?

AT-ATS can…

- import data in fixed format (dictionary-based) and as a comma-separated or tab-separated text file.
- perform all the single-variable statistics associated with the "sum" command. "sum, detail" is also implemented
- do most basic options of the basic "ttest" command
- do the one- and two-way tab command, with the optional "gen" modifier to create an array of dummy variables.
- perform one-way ANOVA
- do Chi-Square tests and other aspects of the "table" command
- do Multivariate OLS regression with the "regress" command and the "predict" command to extract residuals and predicteds. Several commands can follow a "regress" command:
    - "predict" to extract the fitted residuals or the fitted values themselves
    - "test" to make some conjectures about the relationships among fitted betas by generating Wald F-statistics
    - "`exclusionrestrict`" to find out if two regressions, one having a subset of independent variables relative to the other, are different. Generates an F-statistic. (this is not a STATA command, by the way)
    - The "robust" option to generate Heteroskedastistic-Consistent standard error estimates. (Using HC1. or HC0)
- The "`gen`" and "`replace`" commands for creating new variables
- A few other odds and ends for housekeeping such as logging, reading ".do" files, and timing operations. (The latter are interesting if you are, say, writing your own statistics package and are trying to tune it to run faster.)
- Some extensions for simple ".do" file programming, including if, foreach, and while loops, and user (program) variables
- Generation of histograms and scatter plots in various convenient file formats – *if* GNUplot (`http://gnuplot.sourceforge.net/`) is already installed on the machine

Can AT-ATS read STATA ".dta" data files?

No. That would require reverse-engineering a proprietary data format. It's hard work and not very much fun. Instead, AT-ATS will read data right now in three formats: comma separated ASCII files (like the ones you might get from "Notepad", tab separated ascii files (like the ones you might get from Excel if you asked to save as "CSV"), and fixed-position dictionary files (like the ones you might download from IPCSR). All of these are human readable formats that STATA will export if you ask it to, so if someone sent you a STATA .dta file, ask them politely to issue the command:

```
.outsheet using foo.csv, comma
```

and send you the resulting file!

Can I program in AT-ATS like in STATA?

Short answer is "only a little." However, there is some preliminary support for the following constructs: "if", "foreach", "while", and "do". I describe those at the end of this document.

Any programmability I add to AT-ATS will not be like STATA's. Quite simply, STATA's "ado" programming language is of out-of-control. Created by accretion rather than design, it has a completely inconsistent and painful grammar. If you want to do ".ado" programming, use STATA.

If you want to create entirely new modules and commands, however, you can more easily add them to AT-ATS using the language AT-ATS was written in itself: Perl. Perl is the Swiss Army Knife of programming language and is my favorite for Quick and Dirty hacking. If you're curious, there is plenty of information on the Internet about it including literally thousands of already written modules, one of which may already do what you want.

If you are serious about statistical programming in general, take a look at the "R" language[1]. This is a special statistics programming language, available for free as a "GNU" project. It works, it's fast, it's very capable, and it's absolutely free. "R" is probably the main reason that I will not try to bring AT-ATS up to "production quality." Spending too much time trying to copy STATA is lame, and "R" already does it all, and better and faster than I'll ever achieve in my spare time.

Does AT-ATS do anything that STATA doesn't do?

Well… it crashes sometimes. Seriously, there are a couple of things, like a built-in chi-square goodness of fit test called `csgof` (for STATA you need to download it) and, in general, more flexible extraction of the results from one command to be saved in user variables to be referenced in subsequent commands.

AT-ATS has a facility for user variables, which may be better than STATA's. It's easier to use, for certain.

[1] http://www.r-project.org/

In general, when a mathematical expression is called for, like in an "`if`"clause or "`gen`" statement, AT-ATS is much better than STATA. AT-ATS can handle a command like:

```
. gen foo = ((bar+5)/sin(x)+9*ln(sally-bob))
```

Just about anything that Perl understands is game, including calls to Perl functions that you might provide in a separate file.

A nice thing about AT-ATS is that it allocates memory on the fly automatically. There is currently no reason for the "set mem" command that STATA has. (This may change in future releases.) You can type it in if you like, though. AT-ATS will ignore it.

Any more differences between STATA?

I'm sure there are gazillions, but one more comes to mind right now: AT-ATS is totally case-*in*sensitive. That is, it doesn't make a distinction between capital letters and lower case; not in variable names, not in commands, and not even in the values of variables. I think this is a feature, as it eliminates a common from of error, but you may think this is brokenness. It's a matter of opinion, I guess.

STATA's identifier for a missing observation is ".", and in fact STATA has several kinds of "missing" identifiers, which you may or may not find useful. AT-ATS uses "__undef" to mean, well, undefined, or empty, or no data, or incorrect, or did not answer. I don't think it makes much difference to the user, however, except that you will see "__undef" from time to time. Moreover, if you have an empty observation, you may want to code it was "__undef" in Excel or whatever software you are using.

STATA makes a distinction between "string" variables and "numeric" variables. AT-ATS makes that distinction at the observational level. If you have a variable that has numbers for some observations, but strings for others, and you send it to a function that expects numbers, it will just treat the strings as undefined. If you send it to a function that expects strings, it will treat the numbers as strings. (That is, 5 will become "5.") I think it's just a little easier to work with this way.

What other limitations does AT-ATS have?

There are a lot. Here are the biggies, as I see them:

AT-ATS, written in a scripting language, is not frugal with your computer's memory. For small datasets, this is no big deal, but for large ones, it can grind your machine to a halt, and even crash it. I have found that trying to do a multivariate regression with robust analysis will crash my machine, which has 512MB of memory, if there are many more than 100,000 observations and a few variables. However, I usually get away with it.

An annoying limitation (right now) in AT-ATS is that for regressions, it is not capable of detecting collinearity among independent variables and automatically dropping them. Instead, it simply complains with a message stating suspected collinearity. (It may crash in certain circumstances, too. I'm working on that.) This is a shame, and I'd really like to fix it. However, I really don't know how to do so efficiently. Clearly, STATA does it pretty well, so a solution's out there, if I get good enough at linear algebra. In the meantime

- If AT-ATS says you have collinearity, you definitely do. Drop some variables from the regression run and try again
- If AT-ATS crashes during a regression, collinearity might have been the cause. (It also could be running out of memory on a large dataset. Send me the dataset and I'll look into it!)
- If AT-ATS generates an answer where the standard errors are all huge (more than 100,000x the coefficient values, there was likely very a "near" collinearity in some of the independent variables). You will probably want to drop variables from your regression as the standard errors are not going to be very useful for you otherwise.

I've noticed that AT-ATS is slow. Can anything be done about this?

AT-ATS is slow for a couple of reasons. First, and probably most importantly, the algorithms I used are very straightforward, naïve implementations taken straight out of our textbooks. STATA, written (or accreted) over time by people who knew all the tricks, is almost certainly more clever about how it does calculations, taking shortcuts when possible. An example is the calculation of a simple standard deviation. AT-ATS makes two passes through your data. One to add up all the values and calculate the mean, and a second, adding up all the squared value-minus-mean differences to compute the variance. Most statistics programs will do this in a single pass to save time, but it is a little trickier to get right.

Secondly, AT-ATS was written in a programming language called Perl. Perl is a wonderfully rich language that allowed me to create AT-ATS in very little time. The downside, however, is that Perl is an "interpreted" language, which means that the computer figures out how to execute each line of code only as it gets to that line. This means that it is doing some work just to figure out what to do ever time you run the program. This is in contrast to a compiled language, where the program is converted to efficient machine code only once, by the programmer, long before you run it. In an effort to limit the affect of this particular limitation, AT-ATS by default now uses a special compiled C-library for handling matrix math. This pretty much only speeds up OLS-based stuff, though.

AT-ATS crashed on me!

Tell me about it. Seriously. (`david@kellobowitz.net`) Send me the data set, the commands that caused it to crash, and I'll try to figure it out and get a patched version out. This is alpha software, which means that the GSPP class of 2007 is a guinea big (if it chooses to be so). Eventually, the bugs will settle down, and we'll have our own little personalized statistics program that we can distribute for free.

Why doesn't AT-ATS have a fancy Graphical User Interface (GUI) like every other program written since 1978?

AT-ATS now sports a primitive graphical interface. You can also start STATA in text-mode by passing it the '`-g`' option on the command line.

AT-ATS was, in large part, the product of a long, rainy weekend. I was curious to see just what it would take to hammer out the basic structure of a STATA-like piece of software. It was interesting to explore how the various tests we're learning in class would actually be implemented.

Creating a nice GUI (graphical user interface), on the other hand, would take much longer, while simultaneously being less intellectually stimulating. That is to say, it's a lot of code to arrange a lot of buttons on a screen in just the way you want, and it's about as interesting work as you might imagine. It's not a high priority for me, but if people really are using AT-ATS and they want it, I could look at it.

That said, the GUI has some nice features:

- Clicking on a command in the "review" panel automatically re-types the command in the command window
- Clicking on a variable in the "variables' panel types the name of that variable into the command window
- PgUp and PgDown (as well as the up and down arrow keys) scroll back through old commands
- Drop-down menus can be "undocked' from the screen so that they're always visible
- You can click on any command in the results window (highlighted in green) and it will be retyped into your command window
- Any examples from the online help can be clicked on and run. (provided that the variables they rely on are actually loaded)
- Online help is available two ways: with the "`help`" command and from the help menu.

Tutorial

Using AT-ATS is a lot like using STATA, at least if you've gotten into the habit of typing in your commands.

Starting AT-ATS:

You can start AT-ATS just by clicking its icon in whatever folder you have placed it. You may find that your life is a bit easier if you put the AT-ATS.exe file in the same folder as your data. However, if you don't you can always 'cd' to that directory. When AT-ATS starts, it looks like this:

```
           x                    x
           x                    x
  xxxx    xxxx         xxxx    xxxx    xxxxx    Copyright (C) 2006-2008
      x    x               x    x     x     x   David Jacobowitz
  xxxxx    x     xxxx  xxxxx    x      xxx
 x    x    x          x    x    x         xx    Send questions to:
 x    x    x  x       x    x    x  x  x     x     david@kellobowitz.net
  xxxx x    xx         xxxx x    xx    xxxxx

    "Association does not imply causation."

 Version      : 0:11:283
 Build Date   : Thu Apr 17 18:58:36 2008
 Install Dir  : C:/data/projects/atats/install/build/pdl_gui;
 Perl Version : 5.010000
 This software is distributed for free, under the terms of the
 GNU Public License and comes with ABSOLUTELY NO WARRANTY. For
 details, type "help license".

 + Using Perl::PDL matrix routines. (2.4.3)
 + Started on: Thu Apr 17 19:17:51 2008.
 + GNUplot detected. Graphing enabled.
 . Current directory: c:/data/projects/atats/install/build/pdl_gui

Saving debugging log file to err_atats.log.
.
```

Like STATA, the "command prompt" where you can type commands is simply a period.

You can find out what folder you are in by using the "pwd" command:

```
. pwd
> C:/data/projects/atats/
.
```

And you can '`cd`' to the folder you want to be in:

```
. cd c:\data\school\spring_2006\pp240b\ps3
```

```
> Directory now c:/data/school/spring_2006/pp240b/ps3
```

You can always quit AT-ATS with the "`quit`" or "`exit`". They both do the same thing. Like STATA, if your data in ATATS has not been saved or "cleared," AT-ATS will refuse to quit. You will have to be more forceful if you're really done:

```
. quit
no; data in memory would be lost
. quit, clear

 Goodbye!
```

Getting Help

I’ve created a primitive help system as part of AT-ATS. You can get help either of two ways. One way is to type:

```
. help <topic>
```

One nice trick with this is that if you are running the graphical user interface mode (you probably are) then you can simply click on any of the examples in the help and they will be copied directly into your command window.

On the command line, where topic is the name of the command you’re interested in. If you want to see a list of topics, try:

```
. help topics
```

This will splat the help onto the results screen. The graphical user interface also has access to the same help information, and it will appear in a separate window. On the graphical interface, choose the Help->help option in the upper right hand corner. You will see a list of topics. You can click on any of them to see the related help.

If you’re not finding what you’re looking for, you can do a keyword search of the help. Try

```
. search rainy
```

This will return the list of all help articles that contain the word “rainy”.

I know, the help is sparse. Anybody who wants to help write a more detailed help database is welcome to help!

You can also get a list of possible commands with the command:

```
. commands
```

Logging Your Work

It’s a good idea to generate a log file as you work, so that you can see what you did and recreate the steps, if necessary. This is easy enough to do, and works pretty much like STATA:

```
. log using "tutorial.log"
Log opened on tutorial.log
```

If you want to close the log file, just type:

```
. log close
> Closing logfile on “tutorial.log”
```

AT-ATS is a bit less fussy about opening and closing log files than STATA. If the file already exists, AT-ATS will just overwrite it. If you forget to close a log before opening another, AT-ATS will do it for you. If you quit without closing, AT-ATS will also just close it for you.

Log files are useful for just capturing what you did, as well as all of the AT-ATS output. Sometimes, you'll want to re-run everything that you just did, perhaps after changing a command here or there first. AT-ATS has a nice facility to make this easier called "`log2do`". It takes a log file, strips out the AT-ATS results and any errors and generates a clean "`.do`" file that you can run. You can use it like this:

```
. log using my.log
. [… do some stuff …]
. log close
. log2do using my.log
```

Now you have a .do file you can run. You might want to edit it first, perhaps changing something that you had done earlier, or you can simply re-run it verbatim to recreate your workspace the same way it was before:

```
. clear
. do my.do
```

Getting Data into AT-ATS

STATA has a lot of ways to get a data file loaded. AT-ATS only has a few, but they work fine in the majority of circumstances.

The first, is the `insheet` command, which is used to bring in data files formatted as comma delimited text. One source of such files is a spreadsheet, where you can choose to output in "`.csv`" format. If you were to open one of these files in a text editor like "notepad" you would see something like this:

```
earningswk,female,age,age2,edcat1,edcat2,edcat3,edcat4,edcat5
1,1,22,484,0,1,0,0,0
1,0,54,2916,0,0,1,0,0
1,1,53,2809,0,0,0,1,0
1,0,53,2809,0,0,1,0,0
2,1,49,2401,0,0,0,1,0
…
```

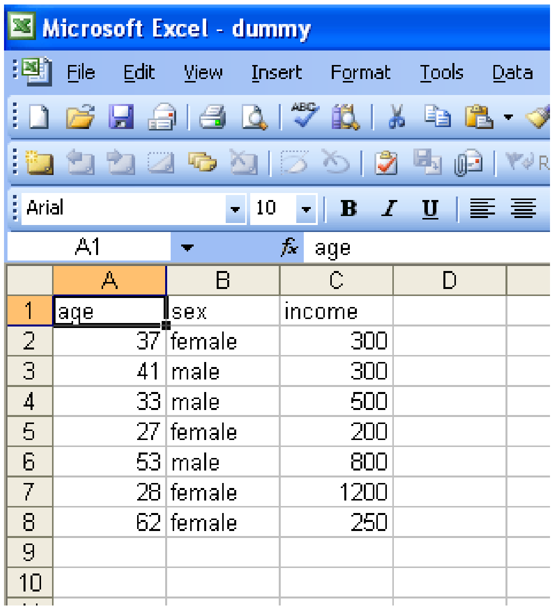
Note that the first line of this file contains the names of the variables. Subsequent lines have the observations.

You can load this file into AT-ATS with the following command:

```
. insheet using ps3data.csv
(9 vars, 12987 obs)
```

See, no problem! No fuss at all.

If the spreadsheet program that you use happens to be Microsoft Excel and you have a working copy installed on your PC, then your life is even easier, because you don't need to mess with .csv files at all. Instead, AT-ATS can read the Excel file (.xls) directly – provided your data is formatted as columns. It might look like this:



	A	B	C	D
1	age	sex	income	
2	37	female	300	
3	41	male	300	
4	33	male	500	
5	27	female	200	
6	53	male	800	
7	28	female	1200	
8	62	female	250	
9				
10				

The command is easy:

```
. insheet using salaries.xls, xls
```

Sometimes your data will not be in a comma delimited format, but in a tab delimited one. These files look like this:

```
earningswk        female  age     age2    edcat1  edcat2  edcat3  edcat4  edcat5
1       1         22      484     0       1       0       0
1       0         54      2916    0       0       1       0
1       1         53      2809    0       0       0       1
1       0         53      2809    0       0       1       0
2       1         49      2401    0       0       0       1
2       1         42      1764    0       0       0       1
…
```

In this case, use the "`, tab`" option to the `insheet` command:

```
. insheet using ps3data.tab, tab
(9 vars, 12987 obs)
```

Second, you may have what is called a fixed-format data file. In these files the variable data is all packed together in a columnar form. These files look like this:

```
007361135000014
007361122500014
007361199999904
007361199999901
007361199999905
…
```

You’ll notice that this data does not have any variable names associated with it. In order to read in a data file like this one, AT-ATS requires an ancillary file, usually called a dictionary. This file tells the names of the variables and in which columns AT-ATS should look for them. Dictionary files are usually produced at the same time as the data file, and they look like this:

```
infix dictionary using ipums.dat {
 year        1-  2
 metaread    3-  6
 gq          7-  7
 valueh      8- 13
 condo      14- 14
 bedrooms   15- 15
}
```

When you want to read in this kind of file into STATA, you need to do a couple of things. First, make sure that the filename mentioned in the dictionary is the name of your data file. Then, simply issue this command:

```
. infix using ipums.dct
(9239 observations read, 6 variables)
```

Finally, you can load a “TRANSPORT” file generate by the statistics program SAS. The SAS website has a description of this format, which I understand is quite common, and is actually the standard interchange format, for the US FDA, for example. (http://support.sas.com/techsup/technote/ts140.html)

The command to load the file is like insheet, but takes no options:

```
inxpt using example.xpt
```

Here’s a trick that you might appreciate, because it saves typing. If you use the `dir` command to get a listing of the current folder’s files, AT-ATS will remember the names of those files, associating them with the number in brackets. Later, you can issue any command that loads or saves a file simply by referencing the number, rather than the typing in the whole name:

```
.dir examp_data
Directory: examp_data

[0] ./
[1] ../
[2] bar.csv
[3] CPS03ps4.csv
[4] ex_file.csv
[5] ex_short.csv
[6] foo.csv
[7] ipums.dat
[8] ipums.dct
[9] ps3data.csv
[10] ps3data.csv.keep
```

```
[11] q.pl
[12] qq.csv
[13] rnd.csv
[14] rnd3.csv
[15] smoking2.csv
[16] test.log

. insheet using 3
(7 vars, 800 obs)
```

What did we do here? I asked AT-ATS to list all the files in the examp_data directory, and then I decided I want to load file number 3. I could have said "`insheet using examp_data/CSP03ps4.csv`", but "`insheet using 3`" was just less effort. Of course, if you're using the graphical interface, this isn't something you'll waste your time on, as you'll just be clicking your way to the file you want.[2]

You can do this for any command that loads or touches a file.

Another thing that AT-ATS will do (which STATA also does) is allow you to download a file directly from the Internet without having to save it to your computer first. This is really just for convenience, especially if you are working with a dataset that changes regularly. Instead of a filename for the "using" argument, just supply an URL instead. For example:

```
. insheet using http://www.ats.ucla.edu/stat/stata/notes/spread.raw
```

loads in an example data file from the UCLA Stata tutorial – which I highly recommend, by the way!

Finally, when you are done manipulating data and creating variables, and want to save the results, you can do so like this:

```
. outsheet using example.csv
```

You can't use an URL with outsheet.

[2] The alert reader will note that this shortcut makes it impossible to work with files whose names are simple numbers. If you had a file that was actually called "3", for example, you might run into problems. This is theoretically a problem, but I, having never in my life named a file "3" am not too worried about it.

In summary:

Input File Format	**Command**
Excel Workbook (.xls)	`insheet using <file>, xls`
Simple comma-delimited text file (.csv)	`insheet using <file or URL>`
Simple tab-delimited text file (.tdf, .txt)	`insheet sing <file or URL>, tab`
Dictionary-based file with numeric data coded in columns. (.dct, .dat)	`infix using <file or URL>`

Output File Format	**Command**
Simple comma-delimited text file (.csv)	`outsheet using <file>`

Merging Datasets

STATA has the "merge" command to combine datasets, using a "key" variables to link them. You can merge datasets in AT-ATS, too, but the command is different and the flexibility is somewhat less. AT-ATS implements a command called `mergesheet` which works something like STATA's marge, but it does so with command-delimited files (`.csv`) rather than STATA's `.dta` files.

It works like ths:

```
. insheet using my_file.csv
. mergesheet ssn using my_other_file.csv
```

The first command, as we are familiar, loads `my_file.csv` into memory. Presume that both `my_file.csv` and `my_other_file.csv` contain, among other variables, a variable in common called "`ssn`". `mergesheet` will simply copy the contents of any observation in `my_other_file.csv` into an observation in `my_file.csv` that has a matching `ssn` variable.

The resulting dataset in memory will have the same number of observations as the first loaded file, `my_file.csv`. Observations that were in `my_other_file.csv` that had key values for `ssn` that did not appear in `my_file.csv` are lost. Observations in `my_file.csv` that had key values for `ssn` that never occurred in `my_other_file.csv` have the new variables from `my_other_file.csv` set to "`__undef`".

Finally, `mergesheet` creates one new variable in your dataset, automatically called "`_merge`". This variable tells you something about how the `mergesheet` operation went for each row. If the row in the original dataset never matched, _merge will be 0. The first observation in the original dataset to match a given key in the new dataset will have _merge = 1. Subsequent matching rows in the original dataset will have higher _merge values.

This is slightly different behavior from STATA, which has some neat controls to enforce the uniqueness of matches. In AT-ATS, if you want to see that there was a unique 1-to-1 match of keys, see that all the _merge values are equal to 1.

Getting an Overview of your Data Set

Probably, the easiest way to get an overview of your data set is to issue the "`sum`" command (which is short for "summary," by the way.) It will give you a summary of basic single-variable statistics for each of the variables in your set.

```
. sum

Variable   |    Obs       Mean  Std. Dev.       Min        Max
-----------+-----------------------------------------------------
       age |  12987  38.914529  11.654080  18.000000  65.000000
      age2 |  12987  1650.1477  930.54456  324.00000  4225.0000
earningswk |  12987  616.69101  452.30472  1.0000000  2884.6100
    edcat1 |  12987  0.0969431  0.2958917  0.00e+000  1.0000000
    edcat2 |  12987  0.3338723  0.4716129  0.00e+000  1.0000000
    edcat3 |  12987  0.2880573  0.4528753  0.00e+000  1.0000000
    edcat4 |  12987  0.1910372  0.3931334  0.00e+000  1.0000000
    edcat5 |  12987  0.0900901  0.2863218  0.00e+000  1.0000000
    female |  12987  0.4939555  0.4999827  0.00e+000  1.0000000
```

You can get a little more info like this:

```
. sum age, detail

                       age
-----------------------------------------------------------
       Obs =      12987
      Mean =  38.914529
  Variance = 135.817582
 Std. Dev. = 11.6540801
  Skewness =  0.0922822
  Kurtosis = 2.1103944

       min =  18.000000
       max =  65.000000

  1st percentile =  18.000000
  5th percentile =  20.000000
 10th percentile =  23.000000
 25th percentile =  30.000000

 median        =  39.000000

 75th percentile =  47.000000
 90th percentile =  55.000000
 95th percentile =  58.000000
 99th percentile =  63.000000
```

You can get a table of all the values the variable takes on like this:

```
. tab age
```

```
        age |      Freq. |    Percent | Cumulative
------------+------------+------------+-----------
         18 |        216 |   1.663202 |   1.663202
         19 |        218 |   1.678602 |   3.341803
         20 |        241 |   1.855702 |   5.197505
         21 |        232 |   1.786402 |   6.983907
         22 |        271 |   2.086702 |   9.070609
         23 |        267 |   2.055902 |  11.126511
         24 |        273 |   2.102102 |  13.228613
         25 |        334 |   2.571803 |  15.800416
         26 |        294 |   2.263802 |  18.064218
         27 |        267 |   2.055902 |  20.120120
         28 |        313 |   2.410102 |  22.530223
         29 |        317 |   2.440902 |  24.971125
         30 |        334 |   2.571803 |  27.542928
         31 |        367 |   2.825903 |  30.368830
         32 |        302 |   2.325402 |  32.694233
         33 |        316 |   2.433202 |  35.127435
         34 |        334 |   2.571803 |  37.699238
         35 |        354 |   2.725803 |  40.425040
         36 |        356 |   2.741203 |  43.166243
         37 |        373 |   2.872103 |  46.038346
         38 |        364 |   2.802803 |  48.841149
         39 |        411 |   3.164703 |  52.005852
         40 |        361 |   2.779703 |  54.785555
         41 |        363 |   2.795103 |  57.580658
         42 |        344 |   2.648803 |  60.229460
         43 |        349 |   2.687303 |  62.916763
         44 |        414 |   3.187803 |  66.104566
         45 |        378 |   2.910603 |  69.015169
         46 |        337 |   2.594903 |  71.610072
         47 |        327 |   2.517903 |  74.127974
         48 |        344 |   2.648803 |  76.776777
         49 |        313 |   2.410102 |  79.186879
         50 |        281 |   2.163702 |  81.350581
         51 |        292 |   2.248402 |  83.598984
         52 |        294 |   2.263802 |  85.862786
         53 |        263 |   2.025102 |  87.887888
         54 |        228 |   1.755602 |  89.643490
         55 |        210 |   1.617002 |  91.260491
         56 |        183 |   1.409101 |  92.669593
         57 |        176 |   1.355201 |  94.024794
         58 |        149 |   1.147301 |  95.172095
         59 |        136 |   1.047201 |  96.219296
         60 |        125 |   0.962501 |  97.181797
         61 |         99 |   0.762301 |  97.944098
         62 |         88 |   0.677601 |  98.621699
         63 |         71 |   0.546701 |  99.168399
         64 |         56 |   0.431200 |  99.599600
         65 |         52 |   0.400400 | 100.000000
------------+------------+------------+-----------
      Total |      12987 |        100
```

You can also see how two variables interact using the "`tab`" command. It's unwieldly and probably pointless if the variables are continuous (and take on lots of values in your data set), but if they're categorical (and don't have too many categories), it can be convenient:

```
. infix using ipums.dct
(9239 observations read, 6 variables)
. tab condo bedrooms


        | condo
        | 0       1       2         (total)
------- + ------- ------- ------- | -------
bedroom |                         |
        |                         |
0       | 426     0       0       | 426
1       | 435     46      12      | 493
2       | 1168    216     72      | 1456
3       | 1206    827     232     | 2265
4       | 526     2128    141     | 2795
5       | 153     1349    17      | 1519
6       | 26      258     1       | 285
------- + ------- ------- ------- | -------
        | 3940    4824    475     | 9239
```

You can also give `tab` the option "`, row`" which will cause it to print out percentages as well as counts, by row. There is currently now related "`col`" function, but I may add later. In the meantime, if you want to see column percentages, consider reversing the rows and the columns in your call to the tab function.

```
. tab condo bedrooms, row

        | condo
        | 0       1       2         (total)
------- + ------- ------- ------- | -------
bedroom |                         |
        |                         |
0       | 426     0       0       | 426
        | 100%    0%      0%      | 100%
1       | 435     46      12      | 493
        | 88.23%  9.33%   2.43%   | 100%
2       | 1168    216     72      | 1456
        | 80.21%  14.83%  4.94%   | 100%
3       | 1206    827     232     | 2265
        | 53.24%  36.51%  10.24%  | 100%
4       | 526     2128    141     | 2795
        | 18.81%  76.13%  5.04%   | 100%
5       | 153     1349    17      | 1519
        | 10.07%  88.8%   1.11%   | 100%
6       | 26      258     1       | 285
        | 9.12%   90.52%  0.35%   | 100%
------- + ------- ------- ------- | -------
        | 3940    4824    475     | 9239
        | 42.64%  55.04%  5.14%   | 100%
```

AT-ATS will not by default cooperate if the horizontally displayed variable has more than seven possible values (seven categories), because that's what fits on the screen without making a mess. You can override this behavior by adding the "`, force`" modifier.

Later, we'll see that we use the same command with the "`chi2`" option to generate a chi-square test for independence.

You can also see each and every one of your data displayed on the screen with the list command. This can splat a lot onto the screen, so you might want to be careful using the command, especially if your dataset is large. The syntax is simple:

```
. list
```

If you want to list only the values of certain variables, the list them by name:

```
. list age sex income
```

If you only want to see a subset of your data, try adding an 'if' clause:

```
. list age income if (age>=18)
```

Working with your Data

Rarely is your data ready to run tests on the moment you import it. Often, you'll have some work to do first, like throwing away observations that are garbage or broken, or just don't make sense. Or you will have to generate some new variables. Perhaps you'll want to create a new categorical variable out of a continuous variable by breaking it into quartiles, quintiles, or some of the kind x-ile.

Dropping entire observations is pretty easy. For example. Let's say we wanted to drop every single observation in a data set that has the valueh variable greater than 100000. Think of this as dropping rows in a spreadsheet.

```
. drop if valueh>100000
8913 observations dropped
```

You can also drop variables. This is just getting rid of variables you don't think are useful.

```
.drop valueh
```

This just got rid of the valueh variable.

STATA's drop can do more stuff, but AT-ATS's is pretty basic.

In addition to dropping variables, you can create new ones, using the "`gen`" command. Say you wanted to create a new variable called oldster, equal to 1 if "age" is greater than 55, and 0 otherwise:

```
. gen oldster = age>55
```

What if you wanted to make a new variable that was age converted into quintiles?

```
. gen age_quintiles = xtile(5;age)
```

Note that the `xtile` function takes two arguments, the number of groups, and the variable. The separator is a semicolon. This is different from STATA. This is an area where the difference can be a pain if you were trying to write, for example, a STATA/AT-ATS compatible ".do" file. However, it made the programming much easier for me.

Also note, the `xtile` function return categories 0 through (n-1), where n is the bincount requested. Also, `xtile` is particularly slow for large datasets. (I'll look into why that is.)

The '`gen`' command can also reference rows in your observation directly. This is useful for creating ID numbers. The row number is always available as "`_n`"

```
. gen ids = _n
```

_n is useful for something else, too. Let's say you have time series data and you want to make some variable references an earlier time or a later time (from other observations). You could do something like this:

```
. gen smoothed_income = (income[_n-1] + income[_n+1]  + income)/3
```

What did we do here? We created a new variable called smoothed_income that is the average of the current value, the previous value, and the next value. Note that for the first and last observation in your dataset, smoothed_income will be "`__undef`". That's life when you're working intertemporally![3]

A handy thing to know is the turnary operator: "? :" It is useful for doing if/then/else comparisons on data. For example:

```
.gen foo = beer ? 1 : wine ? 2 : 0
```

This is equivalent to saying, "is beer true? If so, then foo should be set to 1. Otherwise, is wine true? If so, foo should be set to 2, otherwise, set foo to zero.

Sometimes it's hard to create the variable you want all in one go. For those situations the replace command is handy. It only changes the value of a variable, and only under specific conditions:

```
. gen a_or_b = a + b
. replace a_or_b = 1 if a_or_b==2
```

The "if" clause is required, because without it, this would be the same as the gen command.

A handy thing you can do with the gen command is create variables even when you have no data loaded at all. Why would you want to do this? Because it can be useful for using AT-ATS as a poor man's Monte Carlo environment. For example, you fire up AT-ATS and you want to simulate the interactions of some random variables:

```
. lset obscount = 1000
. gen age = randnorm()*30 + 60
```

(creates 1000 random Gaussian random variates with mean 60 and standard deviation 30)

You can create some more random variables, and then create other variables that interact them, and finally examine that for your results.

Note that once you create your first variable, your data set's depth, and not the value of obscount will determine how many observations future calls to `gen` make; obscount will be ignored.

[3] There is also the magical variable `_ln` as well as `_n`. `_ln` is the the relative row number whenever a subset of the database is in play, like with you are doing a `bysort:` command or if you have specified an `if` modifier. The row referred to by _ln will then be the row within the subgroup

Running Analyses

We’ve had enough creating variables and fooling around. Let’s do some analysis.

T-Tests

Good old basic t-tests. Let’s say you have two variables, one of which is continuous, and the other of which is binary. You want to use the binary variable to separate the samples of the continuous variable into two groups, and then test their differences. This is the simplest of t-tests:

```
. ttest y, by(size)

Two sample t test with equal variances
------------------------------------------------------------------------------
   Group  |   Obs        Mean  Std. Err.  Std. Dev.  [95% Conf. Interval]
----------+-------------------------------------------------------------------
      big |      7  135.71428  6.7778150  17.932412  119.12965  152.29892
    small |      8  137.62500  5.0353660  14.242165  125.71837  149.53162
----------+-------------------------------------------------------------------
 combined |     15  136.73333  4.0017457  15.498694  128.15038  145.31627
----------+-------------------------------------------------------------------
     diff |        -1.9107143  8.3072484            -19.857693  16.036265
------------------------------------------------------------------------------
    diff = mean(big) - mean(small)      t=-0.2300057
    Ho: diff = 0                          degrees of freedom = 13.000000
    Ha: diff < 0                     Ha: diff != 0                 Ha: diff > 0

 Pr(T < t) =  0.4108300   Pr(|T| > |t|) =  0.8216600     Pr(T > t) =  0.5891700
```

You can also test to see if two variables are the same:

```
. ttest b == c
Paired Two sample t test with equal variances
------------------------------------------------------------------------------
   Group  |   Obs        Mean  Std. Err.  Std. Dev.  [95% Conf. Interval]
----------+-------------------------------------------------------------------
        b |     15  65.000000  1.1547005  4.4721360  62.523398  67.476601
        c |     15  278265.00  14699.835  56932.217  246736.79  309793.20
----------+-------------------------------------------------------------------
     diff |     15 -278200.00  14698.682  56927.753 -309725.73 -246674.26
------------------------------------------------------------------------------
    diff = mean(b-c)      t=-18.926866
    Ho: diff = 0                          degrees of freedom = 14.000000
    Ha: diff < 0                     Ha: diff != 0                 Ha: diff > 0

 Pr(T < t) =  1.13e-011   Pr(|T| > |t|) =  2.27e-011     Pr(T > t) =  1.0000000
```

Note that this is automatically a paired t-test, because b and c are both variables in the data set and have one for one correspondence in observations. Think of them, for example, as "before" and "after."

ttest also takes a few arguments, if you like, as options.

You can force AT-ATS not to treat the data as paired:

```
. ttest b == c, unpaired
```

And you can force AT-ATS not to assume that the data have equal variances:

```
. ttest b == c, unequal
```

Finally, whatever your flavor of t-test, you can always limit it to a subset of samples by adding an 'if' clause:

```
. ttest b == c if female
```

Assuming you have a dummy variable called female, this will restrict the comparison of a and b to observations where female was true.

Chi-Square Tests

These are easy. There are two forms of chi-square tests that are currently supported. The first is the test for independence between two variables. This is accomplished by adding the option "`, chi2`" to a tab command that compares two categorical variables:

```
. tab size race, chi2


        | size
        | big     small      (total)
------- + ------- ------- | -------
race    |                 |
        |                 |
asian   | 2       1       | 3
black   | 3       3       | 6
latino  | 0       3       | 3
latinod | 1       0       | 1
white   | 1       1       | 2
------- + ------- ------- | -------
        | 7       8       | 15

       Pearson chi2(4) = 4.28571428571429 Pr=0.36872
```

Be aware that if the variable across the top has more than seven categories, it won't fit on the screen and the output will be suppressed. You'll still get your chi-square value, though.

You can also perform a chi square tests for goodness of fit. In this form, you have to provide the tool with your expectation of how the frequencies should play out. This is done by listing a line of percentages for each category with the command. For example, say you expect the races listed above to have a 20%, 10%, 30%, 20%, 20% distribution:

```
. csgof race, expperc(20 10 30 20 20)

        Chi-square Goodness of Fit

        race    expfreq obsfreq
        ------- ------- -------
        asian         3       3
        black         1.5     6
        latino        4.5     3
        latinod       3       1
        white         3       2

  chisq(4) =  15.666666, p=0.0035007
```

If you don't have a preconceived notion of what the distribution will be, except that they are should be equal, then you can just leave out the expperc() function:

```
. csgof race
```

This will return a similar table with the expected frequencies distributed evenly.

ANOVA

AT-ATS can compute simple one-way ANOVA using the "`oneway`" command or the "`anova`" command. They differ only slightly, in the form of their output. The syntax is:

```
. oneway continuous_var categorical_var
```

or

```
. anova continuous_var categorical_var
```

There should be more than two categories, and there should be more than two observations in each category. If the variance in a given category is zero, the Bartlett's chi-square approximation will be incorrect. I haven't figured out how to handle that case yet.

Here's an example of use, where I am creating a categorical variable of three categories based on age, and then using that to run my ANOVA:

```
. gen ageq = age >20
. replace ageq = 2 if age>40
. replage ageq = 3 if age>60
. oneway earningswk ageq

                        Analysis of Variance
    Source                  SS     df         MS          F    Prob > F
```

```
------------------------------------------------------------------------
Between groups       1.39e+008      2  6.93e+007  357.33974  1.55e-015
Within groups        2.52e+009  12984  193936.22
------------------------------------------------------------------------
Total                2.66e+009  12986  204579.56

Bartlett's test for equal variances:
    chi2(2) =  1069.2686 Prob>chi2 =  0.00e+000
```

Regression

Ah, our old friend Ordinary Least Squares regression. This is the kind of regression you get by default in STATA if you use the regress command, and the syntax is pretty simple:

```
. regress depvar indepvar1 … indepvar2 indepvar3
```

In this case, *depvar* is the name of you dependent variable, and *indepvar1* is the name of your first independent variable, *indepvar2*, the second, etc. You can have as many independent variables as you like, but you must have at least one.

Here's an example:

```
. infix using ipums.dct
. drop if valueh==999999
3883 observations dropped
. gen     actual_bedrooms = bedrooms-1
. replace actual_bedrooms = 0 if actual_bedrooms
. gen iscondo = condo==2
. gen bedrooms_0  = actual_bedrooms==0
. gen bedrooms_1  = actual_bedrooms==1
. gen bedrooms_2  = actual_bedrooms==2
. gen bedrooms_3  = actual_bedrooms==3
. gen bedrooms_4  = actual_bedrooms==4
. gen bedrooms_5m = actual_bedrooms>=5
. regress valueh iscondo actual_bedrooms


     Source |       SS       df       MS        Number of obs = 5356
------------+------------------------------     F(  2, 5353)  = 834.34
      Model |  5.27e+013     2  2.63e+013       Prob > F      = 0.000
   Residual |  1.69e+014  5353  3.16e+010       R-squared     = 0.237648
------------+------------------------------     Adj R-squared = 0.2375588
      Total |  2.22e+014  5355  4.14e+010       Root MSE      = 177634.919

-----------------------------------------------------------------------------
     valueh |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
------------+---------------------------------------------------------------
      _cons |  32209.106 8305.83724  3.8778879 0.00000  15926.342  48491.869
 actual_bed |  98464.752 2557.79428  38.495962 0.00000  93450.452  103479.05
    iscondo | -26210.424 8855.32643 -2.9598485 0.00000 -43570.406 -8850.4424
```

In this example, we just did a regression with two independent variable, iscondo and actual_bedrooms.

If we suspect that there might be some heteroskedasticity in your observations, we can ask AT-ATS to generate robust standard errors:

```
.  regress valueh iscondo bedrooms_1 bedroom_2 bedrooms_3 bedrooms_4
bedrooms_5m, robust

Regression with Robust Standard Errors (HC1)    Number of obs = 5356
                                                F(  6, 5349)  = -1.00
                                                Prob > F      = -1.000
                                                R-squared     = 0.265985
                                                Root MSE      = 6.79e+010

------------------------------------------------------------------------------
     valueh |      Coef.   Std. Err.      t     P>|t|     [95% Conf. Interval]
------------+-----------------------------------------------------------------
      _cons |  192805.12 16583.7419  11.626153 0.00000  160294.35  225315.89
 bedrooms_1 |  5642.5892 18853.2682  0.2992897 0.75362 -31317.357  42602.536
 bedrooms_2 |  30015.180 16971.4309  1.7685710 0.03346 -3255.6126  63285.973
 bedrooms_3 |  107209.89 16842.2845  6.3655198 0.00000  74192.281  140227.51
 bedrooms_4 |  251757.12 17593.6292  14.309562 0.00000  217266.57  286247.68
 bedrooms_5 |  398275.94 22916.8953  17.379140 0.00000  353349.66  443202.22
    iscondo | -30067.301 5674.80062 -5.2983890 0.00000 -41192.180 -18942.422
```

[Note that the F-Value above is "-1." That's should be fixed in the program you're using. I had it set to one during testing – and its even correct for calculations with robust standard errors. It was hard to get right, I assure you!]

If you're wondering, everything else should match STATA to every decimal place displayed.

Speaking of decimal places, you want more, you can play with this command

```
. rval
```

`rval` prints out all the results and some intermediate results from the last command, in a tree form. Usually these results will be calculated to about 15 decimal places. If you need more, the program can be changed easily to provide them, at the expense of speed.

Finally, like just about every command, regress can be combined with "`if`" to restrict the analysis to some subset of your data.

Now that we've run a regression analysis, we might want to test some hypotheses about the values of the fitted betas. We can compare betas to zero, or some other number, or we can compare them to each other. This is done with the "test" command:

```
. test bedrooms_4 = 0
```

Returns the probability that bedrooms_4 could be zero.

```
.test bedrooms_4 = 456
```

Returns the probability that bedrooms_4 is 456.

```
.test bedrooms_4 = bedrooms_5m
```

Returns the probability that the betas for bedrooms_4 and bedrooms_5m are not the same.

Another command for hypothesis testing is called "`exclusionrestrict`." STATA can do this same test, but it has a different command (with a more complex syntax) to do so. `exclusionrestrict` is a way to allow you to compare the difference between two models: an unrestricted with lots of independent variables and a restricted model with some number fewer independent variables. This test answers the question: Is the more complex model any better a predictor than the simpler one?

To use the command, you must first run two regressions, one with all the variables, one with some subset of them. It does not matter which regression you run first, but AT-ATS will check to make sure that one analysis is a perfect subset of the other.

The command is used like this:

```
. regress y b c


     Source |       SS       df       MS        Number of obs = 15
------------+------------------------------     F(  2,    12)  = 12824.51
      Model |  3361.3607     2  1680.6803       Prob > F      = 0.000
   Residual |  1.5726267    12  0.1310522       R-squared     = 0.999532
------------+------------------------------     Adj R-squared = 0.8567420
      Total |  3362.9333    14  240.20952       Root MSE      = 0.3620114

---------------------------------------------------------------------------
          y |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
------------+--------------------------------------------------------------
      _cons |  146.58634 15.8927878  9.2234505 0.00000  111.95913  181.21354
          b | -1.9811389 0.3678933  -5.3850907 0.00010 -2.7827048 -1.1795729
          c |  0.0004274 0.0000289   14.788401 0.00000  0.0003644  0.0004903

. regress y b

     Source |       SS       df       MS        Number of obs = 15
------------+------------------------------     F(  1,    13)  = 1433.02
      Model |  3332.7000     1  3332.7000       Prob > F      = 0.000
   Residual |  30.233333    13  2.3256410       R-squared     = 0.991010
------------+------------------------------     Adj R-squared = 0.9202234
      Total |  3362.9333    14  240.20952       Root MSE      = 1.5250053

---------------------------------------------------------------------------
          y |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
------------+--------------------------------------------------------------
```

```
     _cons | -87.516666 5.9369440  -14.741029 0.00000 -100.34284 -74.690492
         b |  3.4500000 0.0911365   37.855306 0.00000  3.2531087  3.6468913

.exclusionrestrict

 Regression Significance Comparison Test
 ------------------------------------------------------------------
                   Unrestricted    Restricted     Comparison

 dependent var                y             y

 indep var                _cons         _cons
 indep var                    b             b
 indep var                    c

 r-squared            0.9995324     0.9910098

 F_dof_numerator                                             1
 F_dof_denominator                                          12
 F                                                   218.69682
 Prob(F)                                             4.57e-009
```

Here we see that the analysis shows that the more specified model is indeed more predictive than the simpler one.

One more command you might be interested in for regressions is "`predict`" This command creates a new variable (like gen) but populates it with the predicted values from the last regression you ran. It takes an optional argument "resid" that tells it to return the residuals rather than the predicted values.

```
. regress y b c d
. predict foo
```

Now you have a new variable called *foo* that contains the predicted values from your fitted function for *y*.

You can get a variable *bar* that contains the residuals like this:

```
. predict bar, resid
```

You could have done the same thing, by the way, like this:

```
. predict foo
. gen bar = (y-foo)
```

What is this good for? Well, one thing that comes to mind is if you want to do WLS (weighted least squares) to try to correct for heteroskedasticity. You can always turn on "robust" to get hetereskedastistic consistent standard error estimates, but if you think you know the form of the heteroskedasticity, you can also (or instead) try to correct for it with WLS. Under WLS, you weigh each observation by an estimate of it's error. Here's an example (from a PP240B problem set)

```
. regress emplw sex age age2 black othrac edeq12 edgt12, robust
. predict estxb
. gen varest = estxb*(1-estxb)
. gen ivarest = 1/varest
. regress emplw sex age age2 black othrac edeq12 edgt12 [aweight=ivarest],
robust
```

Regression with Removal of Fixed Effects

STATA has some nice commands for estimating (and thereby removing/ignoring) fixed effects from a regression analysis. You can do the same thing in AT-ATS, though the commands are slightly different.

Consider the following command:

```
.tab state, gen(statedummy)
```

This essentially makes a "tab" report of the variable state, but it has the side-effect of creating an array of dummy variables, one for each value that your variable "state" takes on. The names of the dummy variables are what you put inside the parentheses to the "gen" option, but postpended with the value of state.

For example, let's say you had 60 obersrvations, 30 of which were from California, 20 of which were from Oregon, and 10 of which were from Washington. You type:

.tab state

and AT-ATS returns:

```
     state |      Freq. |     Percent | Cumulative
-----------+------------+------------+-----------
        CA |         30 |   50.00000 |   50.00000
        OR |         20 |   33.33333 |   83.33333
        WA |         10 |   16.66667 |   100.0000
```

If you had instead typed:

```
. tab state, gen(statedummy)
```
[4]

AT-ATS would have created three new variables:
statedummy_CA, statedummy_OR, statedummy_WA

[4] This works just like STATA except the way the new variables are named. STATA calls all the new variables statedummy1, statedummy2, statedummy3, etc. AT-ATS actually postpends the value to the name (statedummy_CA, statedummy_WA, statedummy_OR), not just some incrementing counter. I think this is better, but experience may eventually prove otherwise.

These would each be true for the observations in which the value of state was equal to the name of the variable and false otherwise.

This can be a real time saver when creating dummy variables, because if you have many to create, the following can get very tedious:

```
. gen dummy1 = foo==1
. gen dummy2 = foo==2
. gen dummy3 = foo==3
  …
. gen dummy324 = foo==324
```

Okay, so, back to compensating for fixed effects. You've got a bunch of dummy variables that you can throw into your regression model, so you can now just type in:

```
. regress wage education statedummy_CA statedummy_OR statedummy_WA
```

Just a few things to note about this. First of all, the above line would have a collinearity problem since you included *all* variations of state. STATA would drop one for you automatically, but with AT-ATS you're on your own. (If you are lucky, AT-ATS will complain. If you are unlucky it will generate enormous values for the standard errors. If you see this, it's a fair bet you have a collinearity problem in your data.) Also, it should be obvious that wage is our dependent variable, and in this case, education is the variable we're really interested in. The stated dummies are just included to control for. We're really not all that interested in their regression coefficients.

Also, like in the steps above, this can be a lot of typing, too, if you had many dummy variables to include. You can save yourself some trouble by using the AT-ATS "varexpand" command[5]:

```
. regress wage education varexpand(statedummy)
```

What does this do? Basically, AT-ATS finds every variable whose *name* matches "statedummy" and plops that list of variables where the `varexpand` directive had been. It's just like as if you had typed in all the names of the variables you wanted to include. You can use varexpand() anywhere a list of variable names is called for in AT-ATS, by the way, not just for the regression command.

Let's do one example from scratch:

```
. * load the data set
. insheet using examp_data\CPS03ps4.csv
```

[5] This does *not* work just like STATA. In STATA, you would type:
. regress wage education statedummy1-statedummy41
You'd be specifying the range with that dash. STATA is treating 1-41 as a numeric expansion
In AT-ATS, we're typing
. regress wage education varexpand(statedummy)
AT-ATS is not doing a numeric expansion, it is doing a textual *matching* of variables. The underling code uses something called "regular expressions." If you know what those are, you know that they are very powerful, allowing all kinds of sophisticated matching.

```
. * generate the dummy variables…
. tab age, gen(agedmy)

. * drop one of the created variables. It doesn’t matter which
. drop agedmy25

. * run our regression, letting varexpand() type out all those dummy names
. * for us
. regress earnwke sex varexpand(agedmy)

. * we’re done with those dummies. Let’s lose ‘em
. drop agedmy
```

Finally, STATA has the command `xtreg …. ,fe` which will do all of the above in one line, and won’t clutter your screen with the coefficients of variables you weren’t interested in, anyway. Alas, AT-ATS doesn’t have that right now. Seems to be a nicety rather than a must-have, since we can still get the same *results* out of AT-ATS, with about the same amount of typing.

Regression with Instrumental Variables

STATA has the command “`ivreg`” to calculate regression using 2SLS (“Two Stage Least Squares”) and instrumental variables. I have started to implement this in AT-ATS, too, though I have not tested it thoroughly.

There are two ways to do instrumental variables in AT-ATS. One way does not require any special command: you just run the two stages using the regular regression command.

Say you want to run two state regression on a variable called “y” with an exogenous variable “X1” and two endogenous variables “X2” and “X3”. For the endogenous variables, you have to instrumental variable to replace them: “X2i” and “X3i”. You can get 2SLS coefficients like this:

```
. regress X2 X1 X2i X3i
. predict X2hat
. regress X3 X1 X2i X3i
. predict X3hat
. regress y X1 X2hat X3hat
```

That gets you the correct coefficients, but the standard errors will be wrong. You will have to apply a correction to them to make the meaningful. I’m still in the process of trying to figure out what that correction is! The textbooks make it seem trivial, but none of them actually say how to do the correction.

You can do the above either in STATA or in AT-ATS, by the way.

But STATA also has a command “`ivreg`” to make this easier. The above regression can be computed in one step like this:

```
. ivreg y X1 (X2 X3 = X2i X3i)
```

STATA and AT-ATS will output the same coefficients as above, but as an extra bonus, the standard errors and confidence intervals will be correct. STATA can take all the usual regression options for ivreg, but right now AT-ATS only understand the basic form.

AT-ATS's implementation of `ivreg` *does* understand "`robust`" but does not understand `[aweight=xxx]` (at this time).

Generating Graphs

I’ve started on support for generating graphics with AT-ATS. There are lots of caveats and limitations, however. Here are the basics:

- AT-ATS can’t draw graphs itself. It uses another piece of software, also available for free on the Internet, called GNUplot to do this. If you want graphs to work, first get and install a copy of GNUplot on your system.
- AT-ATS only supports histograms and scatter plots right now
- AT-ATS syntax for drawing graphs is different from STATA’s

Histograms

It’s easy to make a histogram:

```
. histogram age [output=my_graph]
```

AT-ATS will use GNUplot to generate a graph called my_graph.emf. This is a file that can be viewed with Windows, or read into Microsoft Word. AT-ATS does not actually show the graph itself; it just creates the graph file.

You can specify some other stuff for graphs. For example, you can control the number of “bins” in your histogram with:

```
. histogram income [output=incomes] [bincount=20]
```

By default AT-ATS finds the highest and lowest values in your set and generates a range of bins to encompass everything. But in some situations your data may have some long tails – some outlier values that are very different from the rest, and which would cause the interest part of your data to appear squished up on your histogram. You can then set your own boundaries for the histogram like this:

```
. histogram income [bincount=20] [minval=0] [maxval=100]
```

Now you’ll get 20 bins each 5 wide going from 0 to 100.

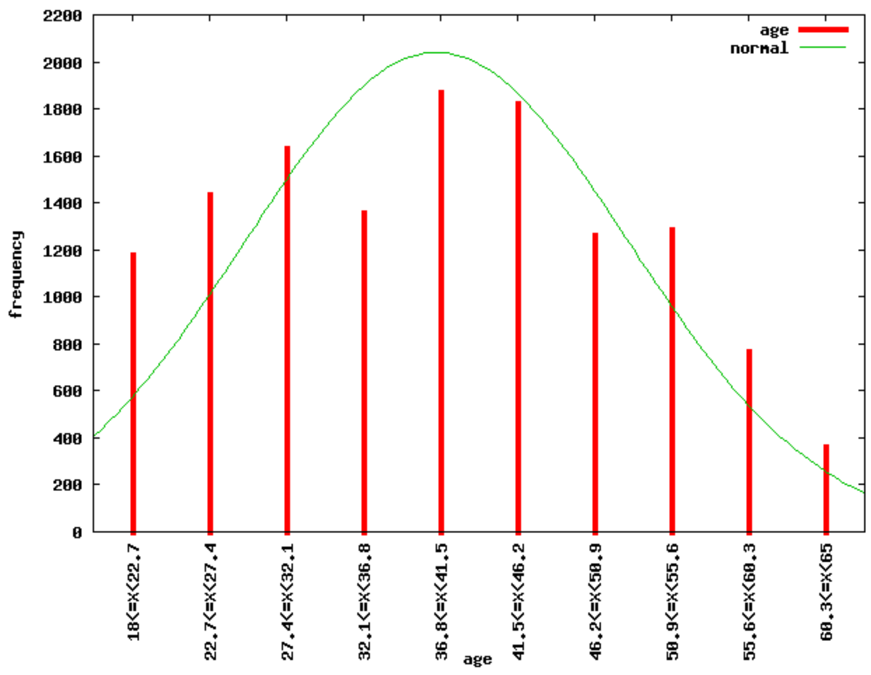
And you can also change the format of the file to various types. For example, by default we generate “.emf” (Enhanced MetaFile files which are a Microsoft Windows graphics format.) But you can also generate a .GIF file or .PNG, or .postscript file just as easily:

```
. histogram income [output=income] [format=postscript]
. histogram income [output=income] [format=png]
. histogram income [output=income] [format=gif]
```

A neat trick that you can do in STATA and AT-ATS alike is to tell it to overlay a normal curve over your data. This helps you to visualize the mean and standard deviation of the data:

```
. histogram age [output=age_graph], normal
```

Here's what the output might look like:



Finally, as always, there is always something that AT-ATS will do that STATA simply refuses to: AT-ATS can make a histogram from categorical data. It will just print the names of the categories under each bar of the graph. Note that in this sense, the "normal" flag is meaningless since categories don't have ordinality.

```
. histogram race [output=racial_makeup], discrete
```

Also note that in this case the number of "bins" or bars is equal to the number of categories; it's not user-adjustable.

Finally, as always, you can restrict the histogram to a subset of your data by adding an if clause. For example, if you have race data encoded as 0=black, 1=white, 2=Hispanic, 3=other, and you want to leave out the "other" you could do:

```
. histogram race [output=racial_wo_other] if (race!=3), discrete
```

Scatter Plots

Scatter plots are pretty easy to generate, too. A two-way scatter plot is generated like this:

```
. scatter age income [output=myplot]
```

Note that income shows up on the Y-axis and age on the X-axis.

You can output all the same formats as the histograms. The default is “emf”.

Here’s a trick you can’t do with STATA – make a 3-D scatter plot!

```
. scatter age education income [output=my3dplot]
```

Not, too bad, eh? By the way, the arguments are in the order of X,Y,Z, so income is on the Z or height axis. There are some fun options for the 3-D plots, too, like “`,color`” and “`,project`”.

Repeating Commands

STATA and AT-ATS both have tricks to cut down on typing. Mostly, these are useful if you want to repeat a command over and over, with only a slight variation.

Here's one that is implemented AT-ATS:
"`bysort:`" Loops

Let's say you had a variable in your dataset that represented citizenship and that "0" meant native born citizen, "1" meant naturalized, "2" meant greencard, etc. You might want to see what the average yearly income was for each of these groups. You could do something like this:

```
. sum inctot if citizen==0
. sum inctot if citizen==1
. sum inctot if citizen==2
```

This quickly gets tedious, especially if you have a lot of categories (like one for every city in the US), and worse, if you miss a category or change categories, your do file may be incorrect. Here's an easier way to do this:

```
. bysort citizen: sum inctot
```

This will run the 'sum' command repeatedly, for each value that citizen takes on.

STATA actually has two variations of this command: "`by`" and "`bysort`". The latter sorts the data by the "by" variable before breaking it into categories. AT-ATS only implements the bysort variation, it doesn't actually sort the data as STATA would. What it does do, however, is make sure that it is grouped correctly. It's a subtle difference.

Say you had data like this

Citizen	income
0	10
0	20
1	15
1	25
0	25
0	10

In STATA, doing just "`by:`" will get you three runs of your command. The first with the first two citizen==0, the second with the next two citizen==1, and a third with the last two of citizen==0. If you did "`bysort:`" then STATA would first sort and you would get only two runs of your command. The first, with all four observations of citizen==0, and the second with the two 1's.

AT-ATS only implements bysort – but it actually does it without sorting your data. You'll get the same result at STATA's bysort except that your data in memory will *not* be sorted. If you want your data sorted, then run the "`sort`" command first.

If you need to do anything fancier than that, you can do so by delving into the next section, "programming!"

Programming (experimental and slightly more advanced stuff)

You can't program in AT-ATS from the command line (right now). All it understands is one command at a time. However, if you are used to doing slightly more sophisticated stuff in ".do" files, AT-ATS does support a little creative programming. You can edit programs in your favorite text editor and then run them in AT-ATS with the "do" command, just as you would in STATA.

The syntax of the language is different from STATA's, which is so twisted that it's not worth recreating.

Local (User) Variables

First, we need to go over the context of local variables in AT-ATS. AT-ATS obviously stores all your sample data in variables as they are named in the input file. Think of these as being vectors of all the observations of that variable. But AT-ATS also has what I call, for lack of a better phrase, local variables. This is a misnomer, because they are in fact global in scope. (Once they are defined any subsequent command can reference them regardless of their nesting in control blocks.) Basically, they're other variables you can set, and for the time being you can only treat them as scalars (that is, individual numbers or strings).

The syntax for setting one of these is:

```
. lset myvar = 5
```

Pretty simple. We created a variable called myvar and we made it equal to five. [One caveat: don't create a local variable with the same name as a variable in your dataset. Subsequently, in certain situations where you are referencing the dataset variable, the local variable can mask it. This would be confusing, to say the least. Also, it should go without saying that you don't want to name variables the same as any commands. I don't check for this right now, but maybe one day I'll add it.]

You can see what local variables are currently set like this:

```
. lval
 myvar = 5
```

So, what are these good for?

`while` Loops

Let's say, now that you want to loop over a few lines of your do-file to do some repetitive task. One way to do it is like this:

```
. while myvar>0 {
  test othervar == myvar
  lset myvar = myvar -1
}
```

This would run the `test` command five times, each time testing to see if `othervar` was equal to the current version of `myvar`.

Right now there is only crude support to dereference the local variables inside the “while” loop. However, you can always textually swap out the *value* for a local variable for its *name* as if the value had been typed thereYou do this by encompassing the variable (or expression of variables) to be swapped out using backticks.

Consider this example:

```
. gen xx0 = some_var
. lset count=1
. while (count<5) {
  gen xx`count` = xx`count-1`*some_var
}
```

Now, what does this do? It does the same as typing in the following:

```
. gen xx0 = some_var
. gen xx1 = xx0*some_var
. gen xx2 = xx1*some_var
. gen xx3 = xx2*some_var
. gen xx4 = xx3*some_var
```

[As an aside, what have we done here? We’ve created a bunch of variables that are copies of some_var raised do increasing powers. We might use these if we were to try to fit data to a polynomial function of some_var rather than just a line.]

What’s inside the backticks is evaluated (as a numerical equation) and the result replaces the original expression. What can be inside the backticks? Basically any calculation the uses local variables, functions, and literal numbers.

There is also no built-in “for” loop, but for the time being you can get the same effect from a “while” loop. I’ll leave that as an exercise for the reader.

`foreach` Loops

There’s another way to loop, too. It’s probably easier when you want to do the same thing to each of a list of variables. It’s a “foreach” loop:

```
. foreach shazzam [ fi fo fum ] {
 regress fee `shazzam`
}
```

This would run three regressions, each time regressing `fee` against each of `fi`, `fo`, and `fum`.

Notice the backticks around `shazzam`. They tell the interpreter to replace the word `shazzam` with the current value of the foreach loop. `shazzam` is just a temporary placeholder for the iterated values in the supplied list.

This scheme for foreach variables is pretty flexible, as it lets you replace any text inside the loop – between the starting and ending brackets.

`if` Blocks

Oh, one other thing: there is also a regular “if” statement. This is different from the “if” that you find in many AT-ATS commands, that lets you filter your observations to just a subset. This if lets you control whether a block of instructions are executed:

```
. if (foo=4) {
 … do some stuff…
}
. if (foo!=4) {
 … do other stuff
}
```