SAS Class Notes
Entering Data

1.0 SAS statements and procs in this unit

libname	An engine to connect to Microsoft files
data	Begins a data step which manipulates datasets
infile	Identifies an external raw data file to read
input	Lists variable names in the input file
datalines	Indicates internal data
set	Reads a SAS data set
proc contents	Contents of a data set
proc print	Prints observations of variables in a data set
proc copy	Copies SAS files from one location to another

2.0 Demonstration and explanation

2.1 Import Wizard, Libnames and Proc import

We will start with inputting an Excel file into SAS first through the SAS Import Wizard.

- **File**
- **Import Data**
- Choose Excel .xls format (this is the default)
- **Click on Next**
- **Click on Browse to select a file: c:\sas_data\hs0.xls**
- The default option is to read variable names from the first line,
- leave as it is.
- **Click on Next**
- Enter a name (**hs0**) for the data set
- **Click on Finish**

Below is the SAS syntax to import the same excel file.

```
proc import datafile="c:\sas_data\hs0.xls" dbms = xls out=hs0;
run;
```

Another way is to use the **libname** statement, which will be reintroduced in a later unit.

```
libname xlsdata 'c:\sas_data\hs0.xls';
proc print data = xlsdata."hs0$"n (obs=10);
run;
libname xlsdata clear;
```

Both of the methods above (menus or syntax) work for other file formats, such as comma-separated or tab-delimited files, and Stata or SPSS datasets. Now we can look at the data or even modify them if we want.

- **Explorer**
- **Libraries**
- **Work**
- **Double click on hs0**
- **Edit**
- **Edit Mode**
- Click on data to modify data

2.2 Creating a library

Creating a library allows us to refer to a file in a specific directory (folder) without typing out the full file path. The command **libname** creates a shortcut that refers back to a specified directory. The two **proc print** commands below that show that you get the same results by either referring to the file name using the library name or the file path.

```
libname in "c:\sas_data\";

proc print data=in.hs1 (obs=10);
  var write read science;
run;
proc print data="c:\sas_data\hs1" (obs=10);
  var write read science;
run;
```

2.3 Data Steps

One of the more commonly used ASCII data formats is the comma-separated-values (.csv) format. Files of this type can be read in through the Import Wizard or **proc import** as shown above, or through a little bit of programming. We will now show how to read in a .csv file with a SAS **data step**. The following segment is the beginning part of the **hs0** file in .csv format. This data file doesn't have variable names on the first line. Also notice that the line in ***bold italics*** has two consecutive commas near the end. This means that there is a missing value in between. In order to read in the data correctly, we use the option **dsd** in the **infile** statement.

```
0,70,4,1,1,general,57,52,41,47,57
1,121,4,2,1,vocational,68,59,53,63,61
0,86,4,3,1,general,44,33,54,58,31
0,141,4,3,1,vocational,63,44,47,53,56
0,172,4,2,1,academic,47,52,57,53,61
0,113,4,2,1,academic,44,52,51,63,61
0,50,3,2,1,general,50,59,42,53,61
0,11,1,2,1,academic,34,46,45,39,36
0,84,4,2,1,general,63,57,54,,51
0,48,3,2,1,academic,57,55,52,50,51
0,75,4,2,1,vocational,60,46,51,53,61
0,60,5,2,1,academic,57,65,51,63,61
```

The following data step will read the data file and name it **temp**. The **infile** statement tells SAS where the location and the name of the ASCII file is. The **input** statement gives the names of the variables in the dataset in the same order as the comma separated file. The **$** after **prgtype** tells SAS that **prgtype** is a string variable, that is, a variable that can contain letters as well as numbers. The **length** statement tells SAS that the variable **prgtype** is a string (as in the **input** statement, the **$** indicates a string variable) and has ten characters (indicated by the **10** following the **$**). By default, SAS allows a string variable to be 8 or fewer characters. If the string is to be longer, you have to tell SAS using the **length** statement. Note that if you have already specified that the variable is a string in the **length** it is not necessary to include the **$** after **prgtype** in the **input** statement; however, doing so is not problematic.

```
data temp;
  infile 'c:\sas_data\hs0.csv' delimiter=',' dsd;
  length prgtype $10;
  input gender id race ses schtyp prgtype $ read write math science socst ;
run;
```

Once we have entered the data, we can list the first ten observations to check that the inputting was successful. Note that **proc print** "prints" the data to the output window, not to a physical printer.

```
proc print data = temp (obs=10);
run;
```

Sometimes we may want to input data directly from within SAS and here is what to do.

```
data hsb10;
  input id female race ses schtype $ prog
        read write math science socst;
datalines;
 147 1 1 3 pub 1 47  62  53  53  61
 108 0 1 2 pub 2 34  33  41  36  36
  18 0 3 2 pub 3 50  33  49  44  36
 153 0 1 2 pub 3 39  31  40  39  51
  50 0 2 2 pub 2 50  59  42  53  61
  51 1 2 1 pub 2 42  36  42  31  39
 102 0 1 1 pub 1 52  41  51  53  56
  57 1 1 2 pub 1 71  65  72  66  56
 160 1 1 2 pub 1 55  65  55  50  61
 136 0 1 2 pub 1 65  59  70  63  51
;
run;

proc print data=hsb10;
run;
```

2.4 Saving SAS Data Files

So far, all the SAS data sets that we have created are temporary. When we quit SAS, all temporary data sets will be gone. To save a SAS data file to disk we can use a data step. The example below saves the dataset **temp** from above as **c:\sas_data\hs0** (SAS will automatically add the file extension .sas7bdat to the file name **hs0**).

```
data 'c:\sas_data\hs0';
  set temp;
run;
```

We can use permanent SAS data files by referring to them by their path and file name.

```
proc print data='c:\sas_data\hs0';
run;
```

To save all the files from one library to another, we can use **proc copy**. For example, we can save all the SAS data files we have created so far in the work library all to another location.

```
libname myout 'h:\sas_data\';
proc copy out=myout in=work;
run;
```

SAS Class Notes
Exploring Data

1.0 SAS statements and procs in this unit

proc contents	Contents of a SAS dataset
proc print	Displays the data
proc means	Descriptive statistics
proc univariate	More descriptive statistics
proc freq	Frequency tables, frequency charts, and crosstabs
ods	Output delivery system, creating output in various formats
proc corr	Correlation matrix and scatterplots
proc sgplot	Produces many types of plots

2.0 Demonstration and explanation

We will begin by submitting **options nocenter** so that the output is left justified. We use the libname statement to refer to a folder of SAS data files. We will continue to use the SAS dataset **hs0** that was created in the previous unit.

Before we start our statistical exploration we will look at the data using **proc contents** and **proc print**.

```
options nocenter;
libname in 'c:\sas_data\';
proc contents position data=in.hs0;
run;
```

```
proc print data=in.hs0 (obs=20);
run;
* If we only want to print some variables, we can use the var statement;
proc print data=in.hs0 (obs=20);
  var gender id race ses schtyp prgtype read;
run;
```

Before we go any further, let's use a data step to make a temporary copy of hs0 and we will still call it **hs0**. Now we can make changes to the temporary data set **hs0**, without making changes to the permanent data set **c:\sas_data\hs0**. If we decide that we want to do so later on, we can save **hs0** as a permanent data set.

```
data hs0;
  set in.hs0;
run;
```

One of the basic descriptive statistics command in SAS is **proc means**. Below we get means for all of the variables. Along with **proc means**, we also show the **proc univariate** output, which displays additional descriptive statistics.

```
proc means data=hs0;
run;

proc univariate data=hs0;
   var read write;
run;
```

With the **var** statement, we can specify which variables we want to analyze. Also, the **n mean median std var** options allow us to indicate which statistics we want computed.

```
proc means data=hs0 n mean median std var;
  var read math science write;
run;
```

We use the **where** statement below to look at just those students with a reading score of 60 or higher.

```
proc means data=hs0 n mean median std var;
  where read>=60;
  var read math science write;
run;
```

With the **class** statement, we get the descriptive statistics broken down by **prgtype** and **ses**.

```
proc means data=hs0 n mean median std var;
  class prgtype ses;
  var read math science write;
run;
```

We can use **proc univariate** to get detailed descriptive statistics for **write** along with a histogram with a normal overlay.

```
proc univariate data=hs0 noprint;
```

```
  var write;
  histogram / normal;
run;
```

We can use **proc sgplot** to get side-by-side boxplots for the variable **write** broken down by the levels of **prgtype.**

```
proc sgplot data=hs0;
  vbox write / category=prgtype;
run;
```

Below we use **proc freq** to get a frequency table for **ses**. The second example uses **proc freq** to produce a bar chart and cumulative frequency graph in addition to the frequency table for **ses**.

```
proc freq data=hs0;
  table ses;
run;

ods graphics on;
proc freq data=hs0;
  table prgtype*ses / plots=freqplot;
run;
ods graphics off;
```

Here we use **proc freq** to get frequencies for **gender**, **schtyp** and **prgtype**, each table shown separately.

```
proc freq data=hs0;
  table gender schtyp prgtype;
run;
```

Below we show how to get a crosstab of **prgtype** by **ses.**

```
proc freq data=hs0;
  table prgtype*ses;
run;
```

proc corr is used to get correlations among variables. By default, **proc corr** uses pairwise deletion for missing observations. If you use the **nomiss** option, **proc corr** uses listwise deletion and omits all observations with missing data on any of the named variables.

```
proc corr data=hs0;
  var write read science;
run;

proc corr data=hs0 nomiss;
  var write read science;
run;
```

In the example below we use **proc corr** to generate a scatterplot matrix. In the second example below, we use **proc sgplot** to get a scatterplot with a confidence ellipse showing the relationship between the two variables **write** and **read**.

```
ods graphics on;
proc corr data=hs0 nomiss plots=matrix;
  var write read science;
run;
ods graphics off;

proc sgplot data = hs0;
  scatter x = read  y = write;
  ellipse x = read  y = write;
run;
```

We can also modify the symbol with the **markerchar** option to use the **id** variable instead of dots. This is especially useful to identify outliers or other interesting observations.

```
proc sgplot data=hs0;
  scatter x=write y=read / markerchar=id;
run;
```

We can also create a scatter plot where we have different symbols depending on the gender of the subjects (using the **group** option). This can be used to check if the relationship between **write** and **math** is linear for each gender group.

```
proc sgplot data=hs0;
  scatter x=write y=read / group=gender;
run;
```

Here are a couple more examples using proc **sgplot**.

```
proc sgplot data=hs0;
  vbar ses /response = write stat=mean limits=both ;
run;

proc sgplot data=hs0;
  histogram read;
  density read / type=normal;
  density read /type = kernel;
run;
```

SAS Class Notes
Modifying Data

1.0 SAS statements and procs in this unit

proc format	Creates formats (aka value labels)
label	Creates labels for variables
rename	Changes the name of a variable in a data step
if then	Executes a statement only if the condition is true
functions	Creating new variables using SAS functions
merge	Merge files

2.0 Demonstration and explanation

2.1 Proc Format, labels and renaming variables

Let's see what the data set looks like. We will continue to use the same data set **hs0** from last unit.

```
libname in 'c:\sas_data\';
proc contents data = in.hs0;
run;
```

Now let's format one of the variables, **schtyp**. To create value labels, we need to use **proc format**. Then we can apply those value labels to the appropriate variable. In the example below, **proc freq** with the **format** statement is used to create a table with the value labels.

```
proc format;
  value scl 1 = "public"
            2 = "private";
run;
proc freq data = in.hs0;
  tables schtyp;
  format schtyp scl.;
run;
```

We can permanently apply a value label to a variable in a data step using the **format** statement. Note that this data step also produces a temporary dataset **hs0b** which is based on the file **c:\sas_data\hs0.**

```
data hs0b;
  set "c:\sas_data\hs0" ;
  * or use set in.hs0;
  format schtyp scl.;
run;
```

In this data step, we will label the dataset and add a variable label to the variable **schtyp**. The **label** option on the **data** statement sets the dataset label. The **label** statement in the data step assigns variable **schtyp** in the dataset **hs0b**.

```
data hs0b(label="High School and Beyond");
  set hs0b;
  label schtyp = "type of school";
  rename gender = female;
run;
```

We will now look at the effects of the data step using **proc contents**.

```
proc contents data = hs0b;
run;
```

In the data step below we change the name of the variable **schtyp** to **public**, and **gender** to **female**. Then we use **proc contents** to see that the changes have been made.

```
data hs0b;
   set hs0b (rename=(schtyp=public gender=female));
run;

proc contents data=hs0b;
run;
```

2.2 Putting things together in a long data step

Now we will run a longer data step to do a variety of tasks. Comments are used to explain the program. (Note the code below replicates some of the code above.)

```
proc format;
  * create value labels for schtyp ;
  value scl 1 = "public"
            2 = "private";

  * create value labels for grade ;
  value abcdf 0 = "F"
              1 = "D"
              2 = "C"
              3 = "B"
              4 = "A";

  * create value labels for female ;
  value fm 1 = "female"
           0 = "male";
run;

* create data file hs1, label it ;
data hs1(label="High School and Beyond") ;

  set in.hs0 (rename=(gender=female));

  * label the variable schtyp ;
  label schtyp = "type of school";

  * apply value labels to schtyp;
```

```
  format schtyp scl.;

  * the if-then statements create a new variable, called prog,
    which is numeric variable ;
  if prgtype = "academic"   then prog = 1;
  if prgtype = "general"    then prog = 2;
  if prgtype = "vocational" then prog = 3;

  * apply value labels to female;
  format female fm.;

  * the if statement recodes values of 5 in the variable race to be missing
(.) ;
  if race = 5 then race = .;

  * create a variable called total that is the sum of read, write, math, and
science ;
  total = read + write + math + science;

  * the if-then statements recode the variable total into the variable grade
;
  if (total < 80) then grade = 0;
  if (80  <= total  <  110) then grade = 1;
  if (110 <= total  <  140) then grade = 2;
  if (140 <= total  <  170) then grade = 3;
  if (total  >= 170) then grade = 4;
  if (total = .) then grade = .;

 * apply value labels to variable grade;
  format grade abcdf.;

run;
```

Let's check to see that everything worked as planned.

```
proc contents data = hs1;
run;
proc print data = hs1 (obs = 20);
run;
proc freq data = hs1;
  tables schtyp*female;
run;
```

Permanently save the dataset as **'c:\sas_data\hs1'**.

```
data in.hs1;
  set hs1;
run;
```

Creating new variables using procedures

There are also a number of SAS procedures and functions that can be used to modify your data. One such procedure is **proc standard**, which can be used to standardize your variables (i.e., give the a mean of 0 and a standard deviation of 1). Below we use **proc standard** to standardize the

variables **read** and **write**. Note that the variables **read** and **write** have been replace by standardized versions of those variables.

```
proc standard data = hs1 mean=0 std=1 out=hs1b;
   var read write ;
run;

proc print data=hs1b (obs=10);
run;
```

2.3 Using SAS functions

In the long data step above, we created the variable **total** by adding the variables **read**, **write**, **math** and **science** together (i.e., **total** = **read** + **write** + **math** + **science**). We could do something similar using the **sum()** function, but there is a major difference between the two methods. That difference is in how missing values are handled. When we add items using "+", a case with missing values on any of the variables listed will have a missing value for the resulting variable. In other words, if a case has valid values of **read**, **write** and **math**, but a missing value for **science**, **total** will be equal to missing for that case. If we use the **sum()** function, any missing values will be treated as though they were zero, and the new variable will be equal to missing only if all of the variables listed are missing. Which method is most appropriate depends on the situation and what you are trying to achieve. The code below shows how to use the **sum()** function in a data step. Together with the **sum** function, we also show the ordinal function, just for the fun of it.

```
data hs1b;
  set hs1;
  total2 = sum(of read write math science);
  * similarly, mean, max, min and more;
  x3 = ordinal(3, read, write, math, science);
run;

proc print data=hs1b (obs=20);
  var read write math science total total2 x3;
run;
```

2.4 Creating group-level variables using proc means

Now we want to create a new set of variables that are equal to the mean of **read, math,** and **write** for each of the three types of programs. First we use **proc means** to create new dataset that contains the mean of **read** by **prog**. Then, we sort the dataset using **proc sort.** Finally, we merge the two datasets (using the **merge** statement in the data step), matching on the variable **prog**. We will discuss the merging process more in the next unit, Managing Data.

```
* creating group-level variables;
* first try;
proc means data = hs1b mean;
   var read math write;
   class prog;
run;
```

```
* second try, it looks good and try to save it to a data set;
proc means data = hs1b;
  var read math write;
  class prog;
  output out = test mean = mean;
run;
proc print data = test;
run;
* third try, fix a couple of things;
proc means data = hs1b nway;
   var read math write;
   class prog ;
   output out = meanscores
         mean =
         max = /autoname;
run;
proc print data = meanscores;
run;
* sort the data;
proc sort data = hs1;
   by prog;
run;
* merge the two data sets, matching on prog and drop extra variables from
readmean;
data merged;
   merge hs1 meanscores ;
   by prog;
run;
proc print data = merged (obs=20);
run;

* another version of it;
proc means data = hs1b nway;
   var read math write;
   class prog ;
   output out = sumstat
   mean(read) = mread
   mean(math) = mmath
   n(read)  = vcase_read
   n(write) = vcase_write;
run;
proc print data = sumstat;
run;
```

SAS Class Notes
Managing Data

1.0 SAS statements and procs in this unit

if and where	Conditional statement
keep	Keeps named variables
drop	Drops named variables
set	Reads in named file(s), append
proc sort	Sorts cases in a dataset
merge	Merges files

2.0 Demonstration and explanation

2.1 Selecting cases using if or where statement

Suppose we wish to analyze just a subset of the **hs1** data file. In fact, we are studying "good readers" and just want to focus on the students who had a reading score of 60 and higher. Here we show how to create subsets based on the criterion of high vs. low reading scores.

```
data highread lowread;
  set in.hs1;
  if read >=60 then output highread;
  if read < 60 then output lowread;
run;

title "high reading scores";
proc means data=highread n mean;
  var read;
run;

title "low reading scores";
proc means data=lowread n mean;
  var read;
run;
* using where statement;
data highread;
  set in.hs0;
  where read >=60;
run;
```

2.2 Keeping variables

Further suppose that our data file had many variables, say 2000 variables, but we only care about just a handful of them, **id**, **female**, **read** and **write**. We can subset our data file to keep just those variables as shown below.

```
data hskept;
  set highread;
  keep id female read write;
run;
```

2.3 Dropping variables

Instead of wanting to keep just a handful of variables, it is possible that we want to get rid of just a handful of variables in our data file. Below we how to remove the variables **ses** and **prog** from the dataset.

```
data hsdropped;
  set highread;
  drop ses prog;
run;
```

2.4 Appending datasets

In this example we start with two datasets, one for males (called **hsmale**) and one for the females (called **hsfemale**). We need to combine these files together to be able to analyze them by gender, as shown below. In this example, we are adding cases, sometimes called "stacking" the data files. We do this by listing both data file names on the **set** statement in data step.

```
title;
proc freq data=in.hsmale;
  tables female;
run;

proc freq data=in.hsfemale;
  tables female;
run;

data in.hsmaster;
  set in.hsmale in.hsfemale;
run;

proc ttest data=in.hsmaster;
  by female;
  var write;
run;
```

2.5 Merging datasets

Again, we have been given two files. However, in this case, we have a file that has the demographic information (called **hsdem**) and a file with the test scores (called **hstest**), and we wish to merge these files together. To merge files together, each file must first be sorted by the same variable. We then save our newly sorted data file as a new temporary file. Both the sorting and the saving can be done with **proc sort**. Next, a data step with the **merge** and **by** statements is used to combine the datasets.

Before we begin, we should look at the data sets.

```
proc print data=in.hsdem ;
run;

proc print data=in.hstest ;
run;
```

Next, we will sort the data sets by the variable that identifies in both datasets, in this case, the variable **id**.

```
proc sort data=in.hsdem out=dem;
  by id;
run;

proc sort data=in.hstest out=test;
  by id;
run;
```

Now we can merge the files and look at the resulting data set.

```
data all;
  merge dem test;
  by id;
run;

proc print data=all;
run;
```

We can see that the two data sets don't match completely. Let's do a better job this time to have more information on the status of merge. We are going to create a variable called merge in the merged data set to indicate where the observation is from. To this end, we make use of a temporary SAS variable "**in**".

```
data all;
  merge dem (in=fromdem) test (in=fromtest);
  by id;
  if fromdem = 1 & fromtest = 1 then merge = 3;
  if fromdem = 1 & fromtest = 0 then merge = 1;
  if fromdem = 0 & fromtest = 1 then merge = 2;
run;
proc freq data = all;
  tables merge;
run;
proc print data = all;
run;
```

SAS Class Notes
Analyzing Data

1.0 SAS statements and procs in this unit

proc ttest	t-tests, including one sample, two sample and paired
proc freq	Used here for chi-squared tests
proc reg	Simple and multiple regression
proc glm	Used here for ANOVA models
proc logistic	Logistic regression
proc npar1way	Non-parametric analyses
proc univariate	Used here for signrank tests

2.0 Demonstration and explanation

Chi-squared test

Below we use **proc freq** to perform a chi-squared test and to show the expected frequencies used to compute the test statistic.

```
proc freq data=in.hs1;
  table prgtype*ses / chisq expected;
run;
```

T-tests

This is the one-sample t-test, testing whether the sample of writing scores was drawn from a population with a mean of 50.

```
proc ttest data=in.hs1 H0=50;
  var write;
run;
```

This is the paired t-test, testing whether or not the mean of **write** equals the mean of **read**.

```
proc ttest data=in.hs1;
  paired write*read;
run;
```

This is the two-sample independent t-test. The output includes the t-test for both equal and unequal variances. The **class** statement is necessary in order to indicate which groups are to be compared.

```
proc ttest data=in.hs1;
```

```
  class female;
  var write;
run;
```

ANOVA

SAS has a procedure called **proc anova**, but it is only used when there are an equal number of observations in each of the ANOVA cells (which is called a balanced design). **proc glm** is a much more general procedure that will work with any balanced or unbalanced design (unbalanced meaning an unequal number of observations in each cell).

In this example we are using **proc glm** to perform a one-way analysis of variance. The **class** statement is used to indicate that **prog** is a categorical variable. We use the **ss3** option to indicate that we are only interested in looking at the Type III sums of squares, which are the sums of squares that are appropriate for an unbalanced design.

```
proc glm data=in.hs1;
  class prog;
  model write=prog / ss3;
run;
quit;
```

Here **proc glm** performs an analysis of covariance (ANCOVA). In this example, **prog** is the categorical predictor and **read** is the continuous covariate.

```
proc glm data=in.hs1;
  class prog;
  model write = read prog / ss3;
run;
quit;
```

Regression

Plain old OLS regression. **proc reg** is a very powerful and versatile procedure. In the following examples we will illustrate just a few of the many uses that **proc reg** has.

```
proc reg data=in.hs1;
  model write = female read;
run;
quit;
```

Specifying **plots=diagnostics** on the **proc reg** statement produces a number of diagnostic graphs. The **output** statement creates a new dataset, called **temp**, which includes the predicted values (by using the **p** = option) and the residuals (by using the **r** = option). The **proc print** displays the values of selected variables from the **temp** dataset.

```
ods graphics on;
proc reg data =in.hs1 plots=diagnostics;
  model math = write socst;
  output out=temp p=predict r=resid;
```

```
run;
quit;
ods graphics off;
proc print data=temp (obs=20);
  var math predict resid;
run;
```

Logistic regression

In order to demonstrate logistic regression, we will create a dichotomous variable called **honcomp** (honors composition), which will be equal to 1 when the logical test of **write** >= 60 is true and equal to zero when it is not true. This variable is created purely for illustration purpose only.

```
data hs2;
  set in.hs1;
  honcomp = (write >= 60);
run;
```

The **proc logistic** performs a logistic regression. It is necessary to include the **descending** option when a variable is coded 0/1 with 1 representing the event whose probability is being modeled. This is needed so that the odds ratios are calculated correctly.

```
proc logistic data=hs2 descending;
  model honcomp = female read;
run;
```

Nonparametric tests

The **signtest** is the nonparametric analog of the single-sample t-test. The **sign** test is part of the output of the tests of location in **proc univariate**. The value that is being tested is specified by the **mu0** option on the **proc univariate** statement.

```
proc univariate data=in.hs1 mu0=50;
  var write;
run;
```

The **signrank** test is the nonparametric analog of the paired t-test. To obtain this test, it is necessary to first compute the difference between the variables to be compared in a separate data step. Then the new difference variable is tested in **proc univariate**. The **signrank** test is found in the section of the output called "tests of location".

```
data hs1c;
  set in.hs1;
  diff = read - write;
run;

proc univariate data=hs1c;
  var diff;
run;
```

The **ranksum** test is the nonparametric analog of the independent two-sample t-test.

```
proc npar1way data=in.hs1;
  class female;
  var write;
run;
```

The **kruskal wallis** test is the nonparametric analog of the one-way ANOVA.

```
proc npar1way data=in.hs1;
  class ses;
  var write;
run;
```