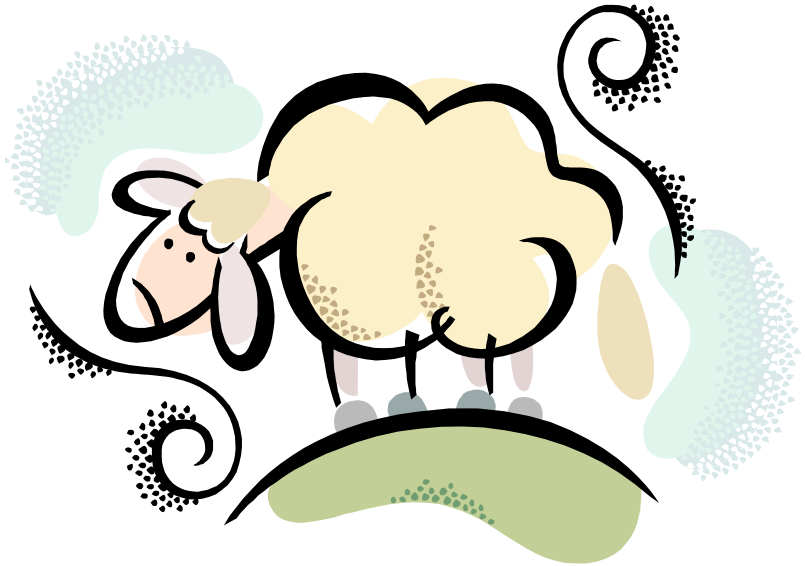


Intermediate SAS

Legal Stuff

This presentation as well as other presentations and documents found on the sheepsqueezers.com website may contain quoted material from outside sources such as books, articles and websites. It is our intention to diligently reference all outside sources. Occasionally, though, a reference may be missed. No copyright infringement whatsoever is intended, and all outside source materials are copyright of their respective author(s).

SAS Lecture Series

An Introduction to SAS Programming	***Intermediate SAS***	*SAS Macros*	*Advanced SAS*

Charting Our Course

- SAS Concepts
- The Data Step – Your First DATA Step
- The Data Step – Basic Data Input using Inline Data
- The Data Step – Saving Your SAS Dataset Permanently (LIBNAME)
- The Data Step – Reading External Data Files using the INFILE, INPUT and
- The Data Step – Dataset Variable Names and Data Types (ATTRIB)
- The Data Step – Creating an External File Using FILE, PUT and FILENAME
- The Data Step – Control Structures (IF-THEN-ELSE, DO END and SELECT)
- The Data Step – About the DATA Statement and OUTPUT/DELETE Statements
- The Data Step – About SAS Formats and Informats
- The Data Step – SAS Expressions and Functions
- The Data Step – Control Structures Redux (Iterative DO, DO WHILE, DO UNTIL)
- The Data Step – Using Arrays in the Data Step (including Temporary)
- The Data Step – Using the SET Statement
- The Data Step – Using the MERGE Statement
- The Data Step – Using *FIRST.* and *LAST.* Temporary Variables
- The Data Step – RETAIN and the Sum Statement (Accumulator Variables)
- The Data Step – More Control Structures (CONTINUE and LEAVE Statements)
- The Data Step – Using DROP=, KEEP= and RENAME=

Charting Our Course *(continued)*

- The Data Step – How SAS Handles Missing Values and the MISSING Statement
- The Data Step – Using the WHERE= Dataset Option
- The Data Step – Using Dates and Times
- The Proc Step – General Information about SAS Procedures
- The Proc Step – PROC PRINT
- The Proc Step – PROC SORT
- The Proc Step – PROC FREQ
- The Proc Step – PROC MEANS
- The Proc Step – PROC FORMAT
- The Proc Step – PROC TRANSPOSE
- The Proc Step – PROC CONTENTS
- The Proc Step – PROC DATASETS
- SAS System Options
- SAS Dataset Options

SAS Concepts

SAS Concepts

According to the *SAS Language Reference: Concepts* manual:

"SAS is a set of solutions for enterprise-wide business users as well as a powerful fourth-generation programming language and an integrated system of software products for performing

- data entry, retrieval, and management
- report writing and graphics
- statistical and mathematical analysis
- business planning, forecasting, and decision support
- operations research and project management
- quality improvement
- applications development.

With base SAS software as the foundation, you can integrate with SAS many SAS business solutions that enable you to perform large scale business functions, such as data warehousing and data mining, human resources management and decision support, financial management and decision support, and others."

In this presentation, we will concentrate on data entry, retrieval and management aspects of SAS as well as some simple report writing. We will concentrate on the `DATA` and `PROC` Steps, both of which are fundamental to the SAS System.

SAS Concepts

The purpose of most computer programs is to *read in data* and *output data*. This can be extended to programs such as:

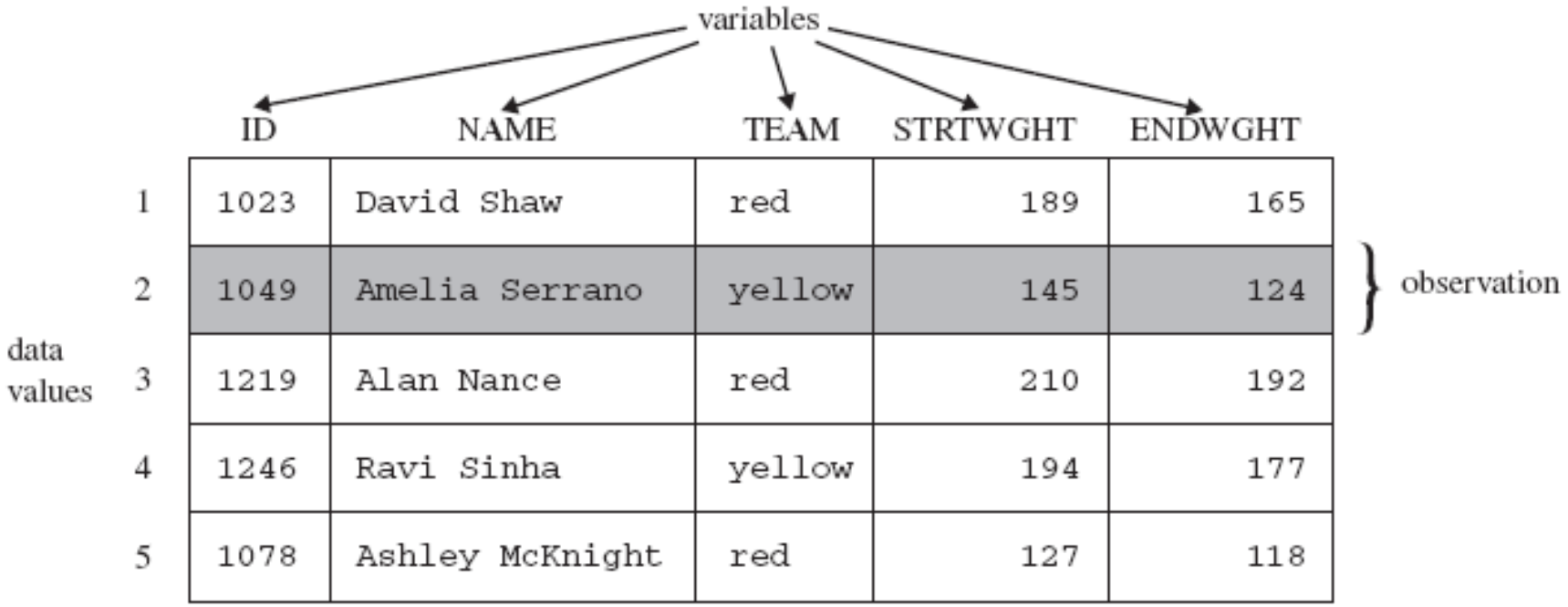
- Video Games – Reads in user's hand movements and outputs the results of those movements to the screen
- Automated Teller Machines – Reads in user's bank card and outputs money or transfer from one account to another
- Credit Cards – Reads in user's credit card and outputs that the bill has been paid

The DATA Step in SAS is no different. It's used to read in external data such as text files, XML files, or SAS proprietary format files (called *SAS datasets*); to manipulate the data by creating new variables; to determine which data to keep based on certain criteria; to evaluate expressions; and to create output files in either the SAS proprietary format (called a *SAS dataset*), or to external files such as text files, etc. to be used by other DATA Steps or by one or more PROC Steps.

The PROC Step takes the data stored in a SAS *dataset* and performs some useful action on that data, whether that be simply printing out the data, summarizing the data, sorting the data, transposing the data (like an Excel Pivot table), or performing a wide variety of statistical analysis on that data. The PROC Step requires that the data provided to it be in the SAS proprietary format called the *SAS dataset*.

SAS Concepts

A *SAS dataset* can be thought of as SAS' version of an Excel spreadsheet or database table and as such houses the data you've read in or created by using one or more DATA and PROC Steps. A SAS dataset (one type of *SAS File*) consists of *rows* and *columns* of data. Other synonyms for rows of data are *observations*, *cases* or *records*. Other synonyms for columns of data are *variables* or *fields*. The most commonly used terminology in SAS is *observations* and *variables*. Each dataset is given a *dataset name* so that you can refer to it later on in the SAS program.



Other types of SAS Files besides datasets are *SAS catalogs* and *SAS Stored Programs*. We will not discuss these in this lecture.

SAS Concepts

Variables in a SAS dataset can be one of two data types: *numeric* or *character*. This is in marked contrast to most database management systems (DBMS's) such as Oracle and SQL Server which have many data types.

A SAS variable with a *numeric data type* can hold numbers such as integers and floating point numbers with decimal points. For example, 12, 76000, 98.6 and 3.14157.

A SAS Variable with a *character data type* can hold text strings. For example, "Les Miserables", "Star Trek: Hidden Frontier", "CIALIS" and "Where did that damn cat go?"

Some computer programming languages require that you define all of your variables with a specific data type. Those types of languages that do require this are called *strongly typed languages*. SAS is a *weakly typed language* meaning that you do not have to define which variable is numeric and which variable is character up front. SAS will determine this from the context in your SAS DATA Step.

Also, some computer programming languages require that you specify the maximum "size" of the data contained in a variable. This is not necessarily required in SAS, but may be warranted. See the next paragraph.

SAS Concepts

As you might have guessed, both numeric and character variables take up *space* in a SAS dataset. As you create *more and more variables* as well as *more and more observations*, the more and more *space* you are using. SAS datasets can become **very** large depending on what you are doing. It's up to the programmer to attempt to create a dataset using as little *space* as possible. This can be achieved by dropping unneeded variables or observations, specifying a maximum size for both numeric and character variables, and by turning on the dataset compression option. Smaller datasets not only take up less space on disk, but can also be processed faster by the SAS System either by another DATA Step or by one of the many PROC Steps.

But, this brings up a point: *Where is a SAS dataset* stored and for how long*?* A SAS dataset can either be *temporary* or *permanent*. *Temporary SAS datasets* only exist for the duration of your SAS session. Once you close SAS, all of your temporary SAS datasets are automatically deleted. *Permanent SAS datasets* are stored in a different location than temporary SAS datasets. You can use the LIBNAME statement to specify a location for your permanent SAS datasets. We talk more about the LIBNAME statement later in the presentation.

SAS Concepts

With each variable in a SAS dataset, you can associate a specific SAS *format* used to let the SAS System know how to **display** the variable's data when using, say, the SAS print procedure PROC PRINT. This format does **not** alter the variable's underlying data and, again, is used just for **display** purposes.

You can even specify a description for each variable in a SAS dataset. This description is called a variable *label* and is used by many SAS procedures in the display of the procedure output.

But, this brings up some important questions such as *How long can my label be?* and *How long can a variable's name be?* and *How many observations can a SAS dataset hold?*

- A SAS dataset name can be up to 32 characters long
- A SAS dataset variable name can be up to 32 characters long
- A SAS label can be up to 256 characters long
- A SAS character variable can hold up to 32,767 characters
- A SAS numeric variable can be up to 8 bytes long (this is the default). We will discuss this in more detail later. This does **not** mean that the maximum number a SAS variable can hold is `99,999,999`!!
- A SAS dataset can hold upwards of `9,223,372,036,854,775,807` observations. Please try not to test this out!

SAS Concepts

SAS procedures usually require the input to be a SAS dataset. Some SAS procedures produce output SAS datasets upon request. These output SAS datasets contain information depending on the procedure you are running, of course, but some examples are frequency counts, summarized data, transposed data, etc.

The basic SAS package – called *SAS Base* – comes with several SAS procedures for doing things like printing datasets, summarizing data, etc. Additional SAS procedures come from purchasing additional SAS packages such as SAS/STAT, SAS/GRAPH, SAS/ETS, SAS/ACCESS, SAS/OR, etc. These packages have several additional SAS procedures you can use in your SAS programs. Documentation for all of these packages as well as SAS Base can be found at SAS' support website: `http://support.sas.com` as well as at sheepsqueezers.com. Please feel free to download these manuals (which are in Adobe Acrobat Reader format) to your personal computer. This way you will always have a copy handy!

Just like other computer software applications, SAS has come out with different versions of its software over the years. Currently, SAS is in version 9.2.

Finally, if you have any problems with your SAS programs, please email your company's SAS Administrator before you contact SAS Institute.

SAS Concepts

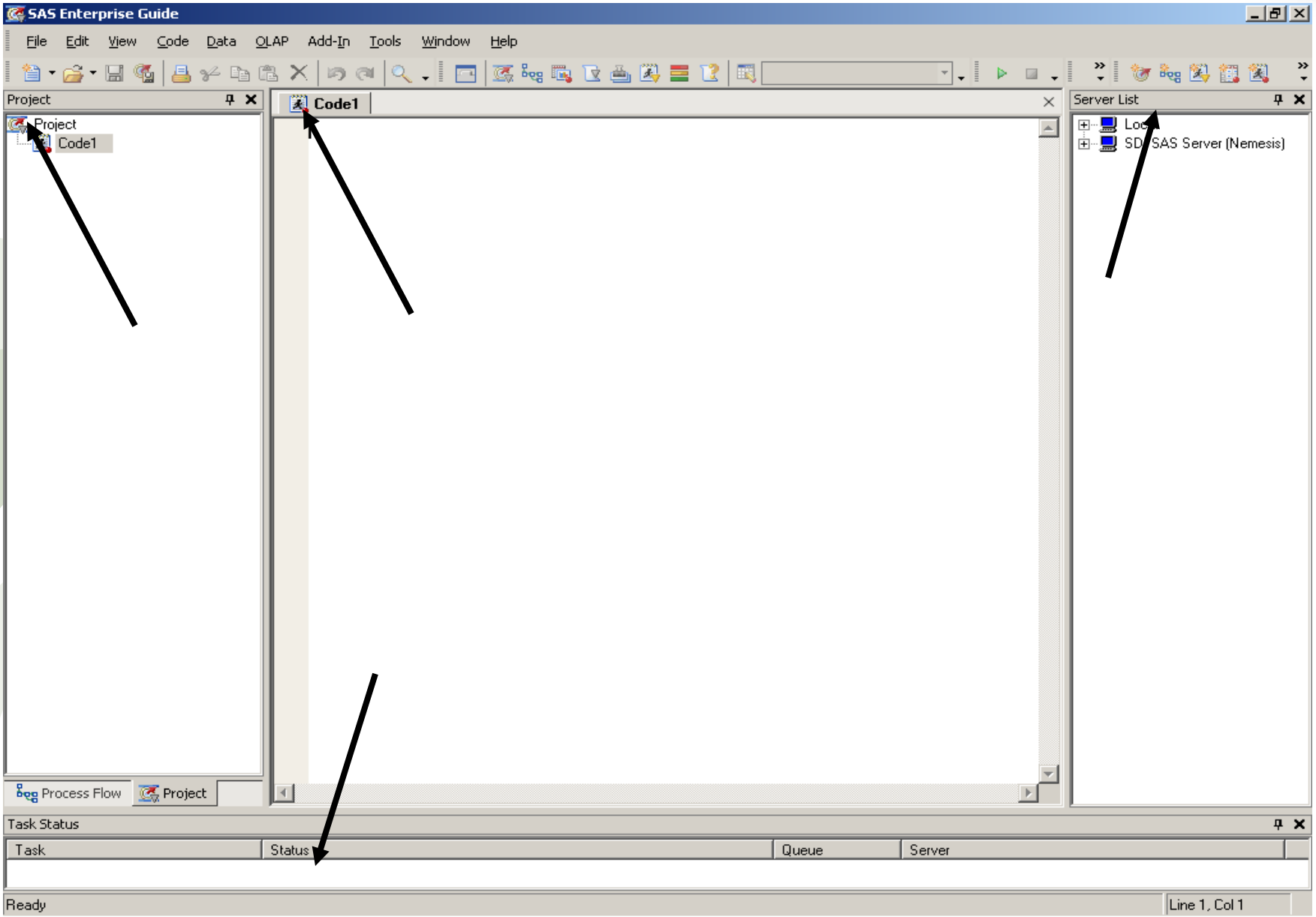
In order to write SAS programs you need either SAS Display Manager or SAS Enterprise Guide. SAS Display Manager is mainly for use by one person and is usually installed on a personal computer. Similar to SAS Display Manager, SAS Enterprise Guide runs on a personal computer as well. But, instead of your SAS programs executing on that same personal computer, SAS Enterprise Guide forwards your SAS programs to a very powerful SAS server to be run. This is much more cost effective than having SAS Display Manager installed on every personal computer in your organization and the server is much more powerful than your little personal computer.

The server where SAS Enterprise Guide runs your SAS programs is called a SAS Enterprise Guide Server.

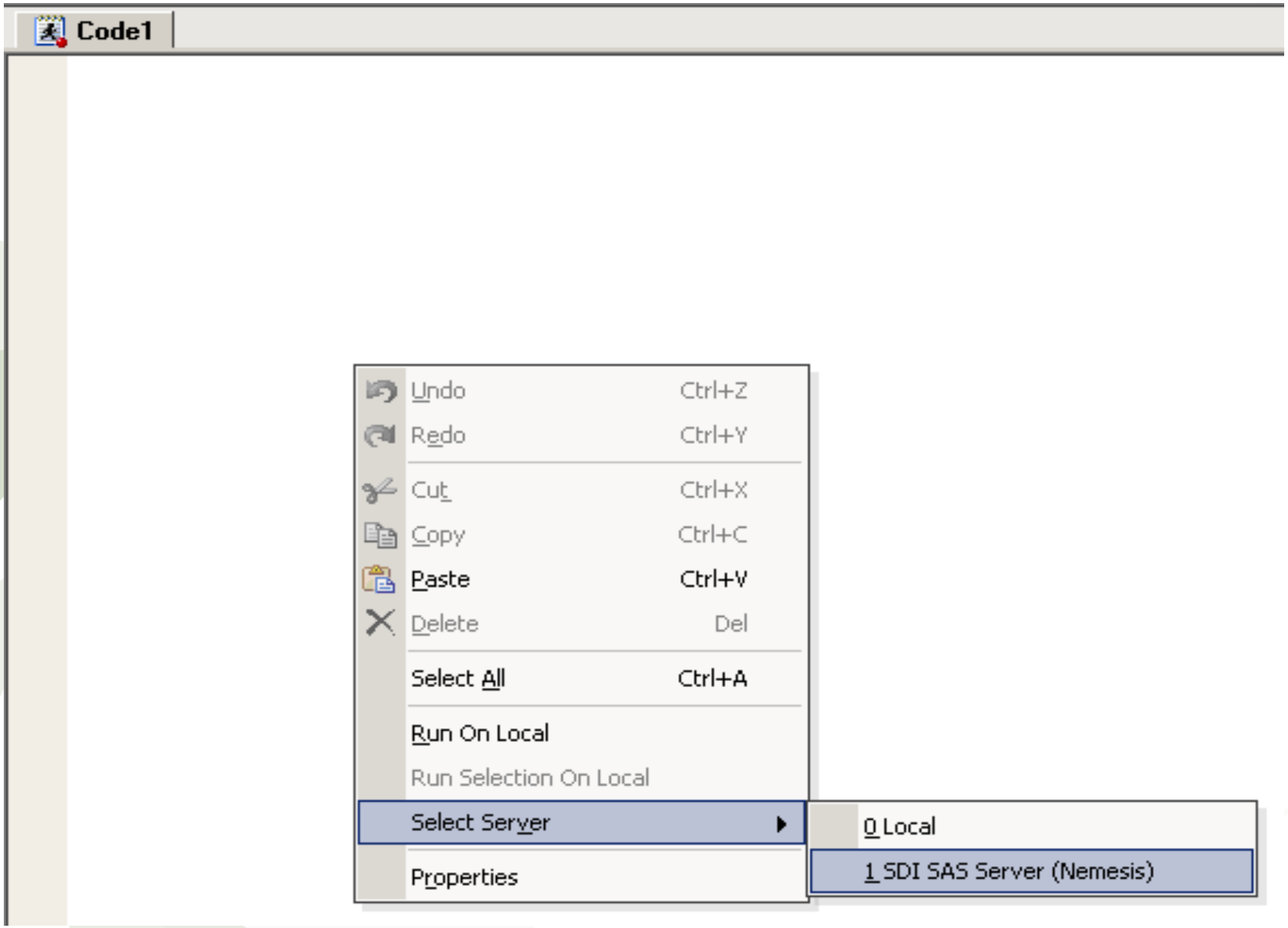
One important fact you must remember about SAS Enterprise Guide is that when you start the software a SAS session (SAS.EXE) is started for **you** specifically on the SAS server. This means that any SAS macro variables, SAS macros, SAS datasets, SAS libnames, SAS filenames, etc. that you submit or create will be available to you within your SAS session and will remain available until either you delete them or close your SAS session.

On the next few slides, we show you the different parts of the SAS Enterprise Guide application software and describe each.

SAS Concepts

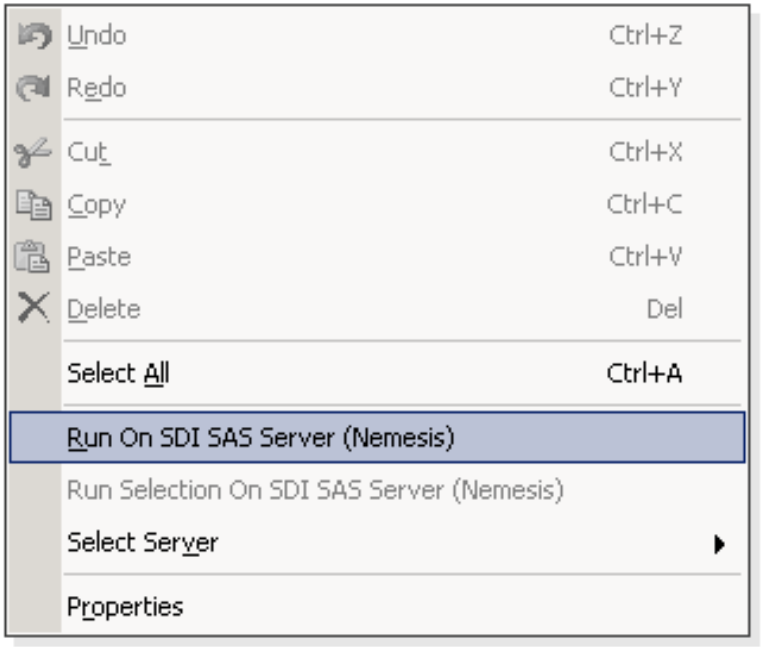


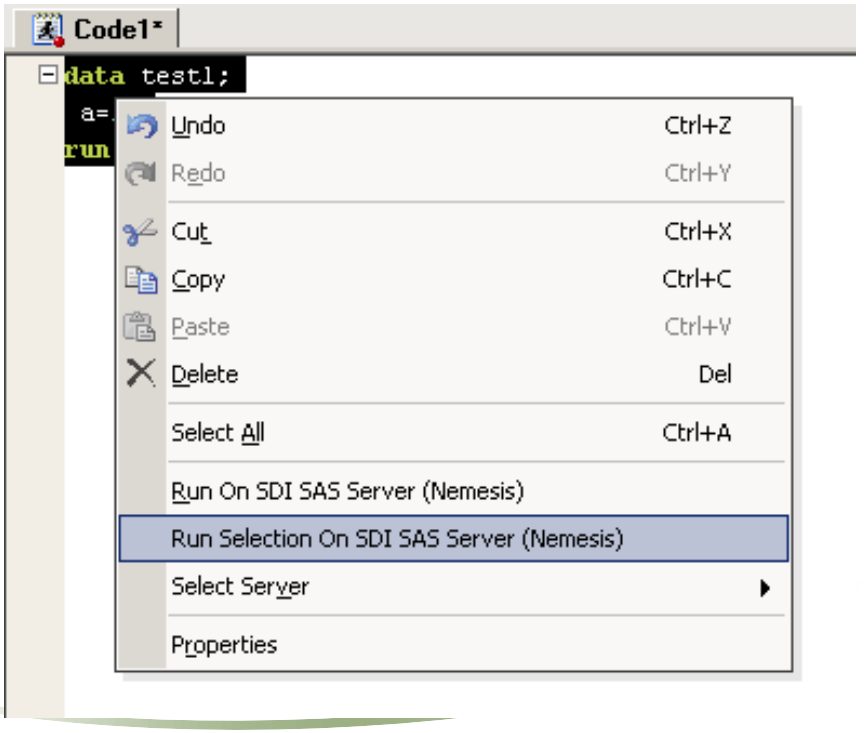
SAS Concepts



SAS Concepts

sheepsqueezers.com





SAS Concepts

Remember that when you submit your SAS programs via SAS Enterprise Guide to your SAS Enterprise Guide server be aware that the *context* has changed. That is, your SAS program is no longer on your personal computer and it doesn't even know about your personal computer. So, any external files you have on your personal computer that you want to use in your SAS program are (almost) inaccessible because your SAS program is running on an Enterprise Guide server. You can think of it like you are sitting at the server console itself running your SAS programs…your personal computer just doesn't enter in to it! If you have external files or SAS datasets stored on your personal computer that you'd like to use in your SAS programs, it's best to copy them to a network location accessible from the SAS Server. We talk more about this later on in the presentation.

You write SAS programs in the *code window* in SAS Display Manager as well as SAS Enterprise Guide. This window uses colors to distinguish between different elements of SAS programming syntax. Once you become used to it, you should be able to determine where a syntax error is just by an incorrect color appearing in the code window.

SAS Concepts

In SAS Enterprise Guide, you can create several code windows by clicking File…New…Code. Each code window contains SAS program statements (code) to accomplish a different task. You may want to create one code window containing all of the SAS macro variables, SAS macros and SAS formats used throughout all of the remaining code windows. The reason for this is that there is only **one** place you need to change your macro variables; the other code windows don't have to be touched (unless there is a syntax error!) since they use these SAS macro variables.

Finally, you can save all of your code windows in SAS Enterprise Guide as a SAS EG *project* by clicking File…Save Project. The project contains all of the SAS code you've written as well as the SAS Logs and Listing files of those code windows you've executed. To save your code in SAS Display Manager, just click File…Save As.

The Data Step

The DATA Step – Your First DATA Step

In this section, we show you how to create a very simple dataset containing one observation and two variables by creating a SAS DATA Step.

```
data MyFirstDataset;
 MyFirstNumericVariable=5;
 MyFirstCharacterVariable="Froddo eats worms!";
run;
```

The SAS code above is a very simple DATA Step. It consists of four lines of code.

The first line consists of the DATA statement followed by the name of the SAS dataset you want to create. Here `MyFirstDataset` is the name of the dataset. At the end of the line is a semi-colon. This semicolon lets SAS know that you've completed the DATA statement.

The second line contains a single *assignment statement*. This assignment statement creates a single variable called `MyFirstNumericVariable` followed by an equal sign followed by the number 5 and again ends in a semicolon. When SAS encounters this, it creates the variable `MyFirstNumericVariable` and assigns it the number 5. Since 5 is a number, the variable `MyFirstNumericVariable` is numeric as opposed to character.

The DATA Step – Your First DATA Step

The third line also contains a single *assignment statement*. This assignment statement creates a single variable called `MyFirstCharacterVariable` followed by an equal sign followed by the text *Froddo eats worms!* in quotation marks and again ends in a semicolon. When SAS encounters this, it creates the variable `MyFirstCharacterVariable` and assigns it the character string "Froddo eats worms!" (excluding the double-quotes). Since *Froddo eats worms!* is a character string, the variable `MyFirstCharacterVariable` is character as opposed to numeric. Since the text *Froddo eats worms!* is 18 characters in length, the variable MyFirstCharacterVariable is set to have a maximum of 18 characters. We'll see later on how to change this.

The fourth line of code tells SAS that you have completed the DATA Step. Note that this is not usually necessary to have, but visually it tells you where the DATA Step starts (with `data`) and ends (with `run`).

When this SAS code is run, SAS creates a temporary SAS dataset called `MyFirstDataset` containing one observation and two variables (`MyFirstNumericVariable` and `MyFirstCharacterVariable`).

Note that once SAS reaches the RUN statement, one observation is created in the SAS dataset consisting of a single numeric variable with the value of 5 and a single character variable with the value *Froddo eats worms!*.

Clearly, we cannot live like this and need a way to read in external data!

The Data Step – Basic Data Input using Inline Data

As we explained in the Concepts section, SAS can read data from external files as well as previously created SAS datasets. SAS can also read data from inline data using the DATALINES statement. Here is an example of reading in 5 lines of data consisting of three variables: PATIENT_KEY, PAT_GENDER and PAT_AGE:

```
data PatientInformation;
 infile datalines;
 input   @1 PATIENT_KEY  4.0
         @6 PAT_GENDER  $1.
         @8 PAT_AGE      2.0;
datalines;
1111 M 29
2222 F 87
3333 F 15
4444 M 42
;
run;
```

As you can see, we start off the DATA Step with the data keyword followed by the SAS dataset name `PatientInformation`, in this case. The INFILE statement on the second line tells SAS **where** the data is coming from, in this case data is coming from data entered in the DATALINES section of the SAS code. Lines 3-5 tell SAS **how** the data in the DATALINES section is to be read in. Note that we start with the INPUT statement on line 3 and we end with a

The Data Step – Basic Data Input using Inline Data

semicolon on line 5. This means that we are entering SAS code on more than one line, but the semicolon tells SAS when we've ended the code. As with most data you read in from external files and inline data, you have to specify the starting column number where the data begins for each variable. In this case, the PATIENT_KEY starts at column 1 (`@1`), PAT_GENDER starts at column (`@6`) and PAT_AGE starts at column 8 (`@8`).

After each starting position, we follow with the name of the variable, in this case PATIENT_KEY, PAT_GENDER and PAT_AGE.

Next, we have to tell SAS how long each variable's data is in the DATALINES as well as the type of data (numeric or character). As you can see, the data for PATIENT_KEY consists of 4 columns (numeric), the data for PAT_GENDER consists of 1 column (character) and the data for PAT_AGE consists of 2 columns (numeric). The syntax `4.0` tells SAS that the data for this variable is located in 4 columns with no decimal places; similar to `2.0` for PAT_AGE. Note that for the character variable PAT_GENDER we have the syntax `$1.` which means that we are reading in a character variable (indicated by the dollar sign) that takes up only one column. The `4.0`, `2.0` and `$1.` are called SAS *informats*.

Next, we have the DATALINES. We start off with the statement DATALINES;, we then enter each observation on a separate row, then we let SAS know that we have finished entering datalines so we end it with a single semicolon.

We follow with a RUN; statement for neatness. At this point we have a temporary SAS dataset called `PatientInformation` consisting of 4 observations.

The Data Step – Saving Your SAS Dataset Permanently

Note that in these first two examples, we've created SAS datasets that are temporary. Recall that temporary SAS datasets last only for the duration of your SAS session and are deleted when you close SAS.

Temporary SAS datasets consist of a single name such as `MyFirstDataset` or `PatientInformation`. These are also called `one-level names`. In order to save a SAS dataset permanently you must use a `two-level name`. A two-level name consists of a library reference followed by a period followed by a dataset name:

```
libref.dsname
```

In order to create a `libref`, you need to use the SAS statement LIBNAME. This statement associates a brief name, called a `libref`, with a specific folder location on disk where your SAS dataset will be stored. The syntax is

```
LIBNAME libref "SAS-data-library";
RUN;
```

LIBREF can be at most 8 characters long and your SAS-data-library (folder) must already exist…SAS will not create a new folder for you:

For example,

```
LIBNAME SDSOUT "C:\MyData";
RUN;
```

The Data Step – Saving Your SAS Dataset Permanently

Note that the libref SDSOUT is now associated with the `C:\MyData` folder. You must specify the LIBNAME statement **before** you are going to use it. Here is a complete example using the PatientInformation DATA Step above:

```
libname SDSOUT "C:\MyData";
run;

data SDSOUT.PatientInformation;
 infile datalines;
 input   @1 PATIENT_KEY  4.0
         @6 PAT_GENDER  $1.
         @8 PAT_AGE      2.0;
datalines;
1111 M 29
2222 F 87
3333 F 15
4444 M 42
;
run;
```

Take note of the DATA statement. In order to store the SAS dataset `PatientInformation` permanently in the `C:\MyData` folder, I prefixed the SAS dataset name with **SDSOUT.**. If you were to take a look in this folder, you would see a file named `PatientInformation.sas7bdat`. This is what a SAS dataset is named on disk.

The Data Step – Saving Your SAS Dataset Permanently

Note that if you would like to remove the association of the libref SDSOUT with the `C:\MyData` folder, you can issue the following statement:

```
LIBNAME SDSOUT clear;
RUN;
```

Here is the complete example:

```
libname SDSOUT "C:\MyData";
run;

data SDSOUT.PatientInformation;
 infile datalines;
 input   @1 PATIENT_KEY  4.0
         @6 PAT_GENDER  $1.
         @8 PAT_AGE      2.0;
datalines;
1111 M 29
2222 F 87
3333 F 15
4444 M 42
;
run;

libname SDSOUT clear;
run;
```

The Data Step – Reading External Data Files

As we've seen, we can read data from inline data within a SAS program. This, of course, can only take us so far! We will now describe how to read in a text file containing rows of data as opposed to inline data.

Just like for reading inline data, we still need to use the INFILE and INPUT statements, but in order to read data from an external file, we also need to use the FILENAME statement. The FILENAME statement associates a brief name, called a fileref (as opposed to a libref with the LIBNAME statement), with a specific file located on disk. Recall that for the LIBNAME statement we associated a libref with a specific **folder** on disk, but the FILENAME statement associates a fileref with a specific **file** on disk.

Let's assume we have a file called C:\MyData\FatKids.dat containing data about fat kids. Here is what the data looks like (the first two rows are for column reference only and do not appear in the text file):

```
00000000011111111
12345678901234567
ALBERT    45   150
ROSEMARY  35   123
TOMMY     78   167
BUDDY     12   189
FARQUAR   76   198
SIMON     87   256
LAUREN    54   876
```

The Data Step – Reading External Data Files

Let's first create the FILENAME statement to associate a fileref with this file:

```
FILENAME TXTIN "C:\MyData\FatKids.dat";
RUN;
```

Next, let's read in this data in a SAS DATA Step using the INFILE and INPUT statements as we did with the inline data:

```
data FatKids;
 infile TXTIN;
 input @1  KidName        $8.
       @10 HeightInInches 2.
       @15 WeightInPounds 3.;
  FattyIndex=WeightInPounds/HeightInInches;
run;
```

Note that we are also creating a new variable called `FattyIndex` using an assignment statement that divides `WeightInPounds` by `HeightInInches`.

As you can see, instead of using the DATALINES keyword on the INFILE statement, we replace it with the name of the fileref, TXTIN in this case.

Similar to the LIBNAME statement, you can also clear a FILENAME statement:

```
FILENAME TXTIN CLEAR;
RUN;
```

The Data Step – Reading External Data Files

The last few examples of the INFILE and INPUT statements have been rather simple and haven't shown off their power. Let's assume that the FatKids data looks like this, where each kid's data is represented by two rows, the first containing the name, height and weight; and the second containing the phone number:

```
00000000011111111
12345678901234567
ALBERT   45   150
(215) 456-1234
ROSEMARY 35   123
(215) 456-2234
TOMMY    78   167
(215) 456-3234
BUDDY    12   189
(215) 456-4234
FARQUAR  76   198
(215) 456-5234
SIMON    87   256
(215) 456-6234
LAUREN   54   876
(215) 456-7234
```

Here is how we can modify the INFILE and INPUT statements to read in this data:

The Data Step – Reading External Data Files


```
data FatKids;
 infile TXTIN N=2;
 input #1 @1  KidName         $8.
          @10 HeightInInches   2.
          @15 WeightInPounds   3.
       #2 @1  PhoneNumber    $14.;
 FattyIndex=WeightInPounds/HeightInInches;
run;
```

Note that `N=2` indicates that there are two incoming rows per one outgoing dataset observation. The `#1` indicates the first line of the `N=2` while `#2` reads the second line of the `N=2`. Note that even though there are 14 lines of data, the SAS dataset `FatKids` will contain 7 rows because 2 incoming rows make up one outgoing observation.

Note that by default, the SAS INFILE statement can read a text file whose row size contains no more than 256 columns across. If you have a text file that exceeds 256 columns, you must specify the LRECL= option on the INFILE statement:

```
infile TXTIN N=2 LRECL=1000;
```

The Data Step – Reading External Data Files

As we mentioned above, the two bolded rows in the data above are there to indicate the column numbers for your reference and are not actually in the external file. Well, what would you do if these two additional rows *were* in the external file? You could open the file up and delete them, but a **real** programmer would use the FIRSTOBS= option on the INFILE statement. This option tells SAS what row number to start with. In this case, if those 2 rows did appear in the text file, we would have to start with row 3:

```
infile TXTIN N=2 LRECL=1000 FIRSTOBS=3
```

Sometimes data within a text file do not appear in nice neat columns. Occasionally, data is delimited by a delimiter like a comma, semicolon, or pipe (|). Here is the FatKids data, but this time it is pipe-delimited:

```
ALBERT|45|150
ROSEMARY|35|123
TOMMY|78|167
BUDDY|12|189
FARQUAR|76|198
SIMON|87|256
LAUREN|54|876
```

Here is the SAS code used to read in this pipe-delimited data:

The Data Step – Reading External Data Files


```
data FatKids;
 infile datalines delimiter="|";
 input KidName        : $8.
       HeightInInches :  2.
       WeightInPounds :  3.;
 FattyIndex=WeightInPounds/HeightInInches;
datalines;
ALBERT|45|150
ROSEMARY|35|123
TOMMY|78|167
BUDDY|12|189
FARQUAR|76|198
SIMON|87|256
LAUREN|54|876
;
run;
```

Take note of the INFILE option DELIMITER="|". This tells SAS that the incoming data is pipe-delimited. You can replace the pipe with a comma, semicolon or whatever delimiter your data has. Also, take note that we removed the column location indicators so that there is no `@1` for KidName. With delimited data, it is not necessary. Also, note that we have placed a colon (:) between the variable name and the informat. This colon tells SAS to use that informat when reading in the data, but takes the width specifier as a maximum.

There's a lot more on this topic…see the SAS manuals for more information!

The Data Step – Dataset Variable Names/Data Types

As we've mentioned before, SAS variables can be one of two data types: numeric or character. As you've seen above, SAS can determine the data type automatically in an assignment statement, or you can give SAS a clue when using an informat while reading inline or external data.

There is a nice SAS statement called ATTRIB which you can use to associate informats, formats, and lengths for each variable in your SAS dataset. Here is an example using the FatKids data:

```
data FatKids;
 attrib KidName        informat=$8. format=$8. length=$8
        HeightInInches informat=2.  format=2.  length=8
        WeightInPounds informat=3.  format=3.  length=8;
 infile datalines delimiter="|";
 input KidName
       HeightInInches
       WeightInPounds;
 FattyIndex=WeightInPounds/HeightInInches;
datalines;
ALBERT|45|150
ROSEMARY|35|123
TOMMY|78|167
BUDDY|12|189
FARQUAR|76|198
SIMON|87|256
LAUREN|54|876
;
run;
```

The Data Step – Dataset Variable Names/Data Types

First note that we used the ATTRIB statement to assign an informat, a format and a length for each variable. Let's ignore the LENGTH= option for now. The informat specified is just the same as the informats on the previous INPUT statements. Note that since we've specified informats in the ATTRIB statement, we don't really need them on the INPUT line, so as you see all we have left are the names of the variables. The FORMAT= option specified a format for printing.

Now, the LENGTH= statement tells SAS how long the variable is and whether the data type is numeric or character. If there is a dollar sign ($), then the variable is character; otherwise, it is numeric. Note that for the two numeric variables we've specified an "8". As mentioned above, this does **not** mean the maximum number is 99,999,999. Here is how LENGTH= works for Windows:

Table 23.1 Significant Digits and Largest Integer by Length for SAS Variables under Windows

Length in Bytes	Largest Integer Represented Exactly	Exponential Notation
3	8,192	2^{13}
4	2,097,152	2^{21}
5	536,870,912	2^{29}
6	137,438,953,472	2^{37}
7	35,184,372,088,832	2^{45}
8	9,007,199,254,740,992	2^{53}

For example, if you know that a numeric variable always has values between 0 and 100, you can use a length of 3 to store the number and thus save space in your data set.

The Data Step – Dataset Variable Names/Data Types

So, if you specify LENGTH=3, then this numeric variable can take a maximum integer value of 8,192. By default, SAS assigns a LENGTH of 8. If your variable will contains decimal places, **always** specify a LENGTH of 8!! For both HeightInInches and WeightInPounds, we could use LENGTH=3.

You can also assign a label for each variable using the ATTRIB statement:

```
ATTRIB HeightInInches INFORMAT=2. FORMAT=2. LENGTH=3
                      LABEL="Height of oinker in inches";
```

Note that instead of assigning an informat, format, length and label using the ATTRIB statement, you could assign each one separately by using the corresponding individual statements: INFORMAT, FORMAT, LENGTH, LABEL:

```
data FatKids;
 length KidName $ 8 HeightInInches 8 WeightInPounds 8;
 informat KidName $8. HeightInInches 2. WeightInPounds 3.;
 format KidName $8. HeightInInches 2. WeightInPounds 3.;
 label HeightInInches="Height of oinker in inches";
 infile datalines delimiter="|";
 input KidName
       HeightInInches
       WeightInPounds;
 FattyIndex=WeightInPounds/HeightInInches;
datalines;
…
;
run;
```

The Data Step – Creating an External File

Just as we can read in data from an external file into our SAS dataset, we can also write out data to an external file from data in our SAS dataset. Recall that when we read in data from an external file we used the FILENAME, INPUT and INFILE statements along with INFORMATs to tell SAS how to read in data from the external file.

To write out data to an external file, we will use the FILENAME, PUT, FILE statements along with FORMATs to tell SAS how to write out data to the external file.

Take note of the correspondence! The FILE statement tells SAS which fileref – as specified in a FILENAME statement – to write data to. The PUT statement tells SAS which columns to use as output.

First, here is a simple example:

```
filename OUTTXT "C:\MyData\Tiny.txt";
run;

data _null_;
 file OUTTXT notitles noprint;
 MyVar1=1234; MyVar2="Froddo";
 put @1  MyVar1   4.
     @10 MyVar2 $10.;
run;

filename OUTTXT clear;
run;
```

The Data Step – Creating an External File

Here is a more complex example:

```
filename TXTIN "C:\MyData\Fatty.txt";
run;

filename TXTOUT "C:\MyData\Fatty2.txt";
run;

data _null_;
 attrib KidName        informat=$8. format=$8.    length=$8
        HeightInInches informat=2.  format=2.     length=8
        WeightInPounds informat=3.  format=3.     length=8
        FattyIndex                  format=12.4   length=8;
 infile TXTIN delimiter="|";
 file TXTOUT notitles noprint delimiter=":";
 input KidName
       HeightInInches
       WeightInPounds;
 FattyIndex=WeightInPounds/HeightInInches;
 put KidName
     HeightInInches
     WeightInPounds
     FattyIndex;
run;

filename TXTIN clear;
run;

filename TXTOUT clear;
run;
```

The Data Step – Creating an External File

Here is what the data looks like in Fatty2.txt:

```
ALBERT:45:150:3.3333
ROSEMARY:35:123:3.5143
TOMMY:78:167:2.1410
BUDDY:12:189:15.7500
FARQUAR:76:198:2.6053
SIMON:87:256:2.9425
LAUREN:54:876:16.2222
```

Note that if you want to **add** data to a pre-existing file, you have to use the `MOD` option on the FILE statement; otherwise, your file will be truncated first:

```
data _null_;
 attrib KidName        informat=$8. format=$8.    length=$8
        HeightInInches informat=2.  format=2.     length=8
        WeightInPounds informat=3.  format=3.     length=8
        FattyIndex                  format=12.4   length=8;
 infile TXTIN delimiter="|";
 file TXTOUT notitles noprint delimiter=":" MOD;
…
run;
```

The Data Step – Control Structures

Computer programs are inherently stupid. You, the programmer, have to help your program make decisions. One way to do this is to graft a part of your brain to the motherboard of your computer. A less drastic way is to learn how to use SAS' control structures such as IF-THEN-ELSE, DO END, Iterative DO, DO WHILE, DO UNTIL, and SELECT. In this section, we concentrate on IF-THEN-ELSE, DO END and SELECT. We will return with the remaining control structures later on.

IF-THEN-ELSE

You use the IF-THEN-ELSE control structure to execute certain portions of your code depending on certain criteria. Below is an example including DO END,

```
data FatKids;
infile datalines delimiter="|";
input KidName        : $8.
      HeightInInches : 2.
      WeightInPounds : 3.;
 if KidName="ALBERT" or KidName="TOMMY" or KidName="BUDDY" or KidName="SIMON" then do;
  FattyIndex=2.5*WeightInPounds/HeightInInches;
 end;
 else if KidName="ROSEMARY" or KidName="LAUREN" then do;
  FattyIndex=0.5*WeightInPounds/HeightInInches;
 end;
 else do;
  FattyIndex=WeightInPounds/HeightInInches;
 end;
datalines;
```

The Data Step – Control Structures

Subsetting IF Statement

One variation on the IF-THEN-ELSE Statement is the Subsetting If Statement. This statement behaves like a WHERE Clause in SQL, allowing you to subset the data based on certain criteria. For example, we use a Subsetting If Statement below to limit our dataset to fat kids over 200 pounds:

```
data FatKids;
infile datalines delimiter="|";
input KidName        : $8.
      HeightInInches : 2.
      WeightInPounds : 3.;
 if WeightInPounds>=200;
datalines;

…
run;
```

Note that there is no THEN or ELSE statement used with a Subsetting If Statement.

The Data Step – Control Structures

SELECT

Instead of using IF-THEN-ELSE's all over the place, you can use the SELECT statement to make your code more readable. There are two types of SELECT, one which does not include an expression after the SELECT statement (see the example below), and the other one which does include an expression after the SELECT.

```
data FatKids;
infile datalines delimiter="|";
input KidName        : $8.
      HeightInInches : 2.
      WeightInPounds : 3.;
 select;
  when (KidName="ALBERT" or KidName="TOMMY" or KidName="BUDDY" or KidName="SIMON") do;
   FattyIndex=2.5*WeightInPounds/HeightInInches;
  end;
  when (KidName="ROSEMARY" or KidName="LAUREN") do;
   FattyIndex=0.5*WeightInPounds/HeightInInches;
  end;
  otherwise do;
   FattyIndex=WeightInPounds/HeightInInches;
  end;
 end;
datalines;

…
```

The Data Step – Control Structures

SELECT (continued)

The example SELECT below does include an expression after the SELECT:

```
data FatKids;
infile datalines delimiter="|";
input KidName        : $8.
      HeightInInches : 2.
      WeightInPounds : 3.;
 select (KidName);
  when ('FARQUAR') do;
   FattyIndex=WeightInPounds/HeightInInches;
  end;
  otherwise do;
   FattyIndex=1000*WeightInPounds/HeightInInches;
  end;
 end;
datalines;
…
```

The Data Step – About the DATA Statement

So far we've seen several examples of the DATA Step. Each DATA Step involves specifying the DATA statement followed by a dataset name. In actuality, you can specify **multiple** dataset names after the DATA statement, each one can either contain the same data or you can use the control structures to output certain data to specific datasets. For example, let's output all the fat boys to the FATBOYZ dataset, and output all the fat girls to the FATGIRLZ dataset.

```
data FatBoyz
     FatGirlz
     FatHermafs;
infile datalines delimiter="|";
input KidName        : $8.
      HeightInInches : 2.
      WeightInPounds : 3.;
 select;
  when (KidName="ALBERT" or KidName="TOMMY" or KidName="BUDDY" or KidName="SIMON") do;
   output FatBoyz;
  end;
  when (KidName="ROSEMARY" or KidName="LAUREN") do;
   output FatGirlz;
  end;
  otherwise do;
   output FatHermafs;
  end;
 end;
datalines;

...
```

The Data Step – About the DATA Statement

Notice how each dataset is named after the DATA statement. Also, notice that in order to output the data to a *specific* dataset, you must use the OUTPUT statement followed by one (or more) datasets you want to be the recipient of that observation.

Note that it's possible, based on the IF-THEN-ELSE or SELECT criteria that one or more datasets receive *no* data. This is okay and SAS does **not** signal an error condition if a dataset winds up being empty.

We've seen how to OUTPUT data, you occasionally might need to delete one or more rows of data from a dataset. In that case, you can use the DELETE statement, like this (FARQUAR does not appear in either dataset):

```
data FatBoyz
     FatGirlz;
infile datalines delimiter="|";
input KidName        : $8.
      HeightInInches : 2.
      WeightInPounds : 3.;
select;
 when (KidName="ALBERT" or KidName="TOMMY" or KidName="BUDDY" or KidName="SIMON") do;
  output FatBoyz;
 end;
 when (KidName="ROSEMARY" or KidName="LAUREN") do;
  output FatGirlz;
 end;
 otherwise do;
  delete;
 end;
end;
datalines;
```

The Data Step – About SAS Formats and Informats

As we've seen, you can use the ATTRIB statement to associate a format, informat, length and label with a specific variable. Let's talk more about formats and informats.

A SAS Format is a way to associate a SAS variable with a specific method of outputting that variable's data when used in a SAS procedure which displays that variable. As we will see below, the SAS procedure PROC PRINT prints out data either using the associated formats, a default set of formats, or an override of the formats within PROC PRINT itself. SAS has **a lot** of built-in formats, some of which are:

- `comma`*`w.d`* – this displays a numeric value with a width of *w* characters, followed by *d* decimal places, but also includes commas in the output.
- `dollar`*`w.d`* – this displays a numeric value with a width of *w* characters, followed by *d* decimal places, but also includes commas as well as a dollar sign in the output.

```
data MyDataset;
 format MyVar1 comma14.2 MyVar2 dollar15.2;
 MyVar1=123456789.12;
 MyVar2=234567890.23;
run;

proc print data=MyDataset;
run;
```

Obs	MyVar1	MyVar2
1	123,456,789.12	$234,567,890.23

The Data Step – About SAS Formats and Informats

A SAS Informat is a way to associate a SAS variable with a specific method of inputting a data value when using the INPUT statement. As we've seen above, when we used the ATTRIB statement to associate informats with variables, there was no need to specify informats on the INPUT statement.

SAS has a lot of informats, here are a few:

- $*w*. – inputs a character string of width *w*.
- $char*w*. – inputs a character of width *w*, but preserves leading and trailing blanks
- $upcase*w*. – inputs a character of width *w*, but upcases all of the characters
- comma*w.d* – inputs a character of width *w* containing commas and converts it to a number optionally moving the decimal place over to the left *d* places
- *w.d* – inputs a character of width *w* and *d* decimal places and converts it to a number

```
data MyDataset;
 infile datalines;
 input @1  MyVar1      4.1
       @8  MyVar2 $char6.
       @15 MyVar3 comma8.2;
datalines;
1234   ABCDEF 1,234.12
;
run;
```

The Data Step – SAS Expressions and Functions

So far, the only new variable we have created is the FattyIndex from the FatKids dataset. This variable was created by the evaluation of the following expression:

```
FattyIndex=WeightInPounds/HeightInInches;
```

The standard arithmetic operators apply in expressions like this: + (Addition), - (Subtraction), * (Multiplication), / (Division), and ** (Exponentiation). You can also use as many parentheses as needed to make the expression clear.

For example, we can *pimp out* the FattyIndex like this:

```
FattyIndex=5*(100+WeightInPounds)/(HeightInInches**2.5);
```

SAS also provides a lot of functions you can use in your expressions, such as:

- `ABS(x)` – returns the absolute value of x
- `EXP(x)` – returns the exponentiation of x (e to the x power)
- `FACT(x)` – returns the factorial of x
- `INT(x)` – returns the integer portion of x
- `FLOOR(x)` – returns the largest integer that is less than or equal to x
- `CEIL(x)` – returns the smallest integer that is greater than or equal to x

Here is a *tricked out* FattyIndex:

```
FattyIndex=INT(FACT(5)*(100+EXP(WeightInPounds))/(HeightInInches**2.5));
```

The Data Step – SAS Expressions and Functions

The functions listed above all take one variable, but SAS has many functions that can take two or more variables. For example, if you want to sum up several variables within an observation, you can use the SUM() function:

```
CrazySum=sum(HeightInInches,WeightInPounts);
```

Notice that, in this case, the sum function is using two variables to compute the sum. You can enter as many variables as you want:

```
AllSum=sum(VAR1,VAR2,VAR3,VAR4,VAR5,VAR6);
```

There are many functions that take two or more variables:

- `SUM(v1,v2,…)` – returns the sum of all of the listed variables.
- `COMB(n,r)` – returns the number of combinations of n elements taken r at a time
- `PERM(n,r)` – returns the number of permutations of n elements taken r at a time
- `ROUND(x,u)` – returns x rounded to u units
- `MEAN(v1,v2,…)` – returns the average (mean) of all of the listed variables

```
AllMean=mean(VAR1,VAR2,VAR3,VAR4,VAR5,VAR6);
```

Note that entering in all of those variables in a function can be a pain, so SAS created several ways to avoid this.

The Data Step – SAS Expressions and Functions

If you have a lot of variables in your SAS dataset, instead of listing each one separately in data step functions or in SAS procedures, you can use a SAS *variable list*. There are 4 types of variable lists:

1. Numbered Range Lists
2. Name Range Lists
3. Name Prefix Lists
4. Special name lists

Numbered Range Lists

Let's assume that you have several variables in your SAS dataset named like `MON1`, `MON2`, `MON3`, `MON4`, …, `MON36`. If you want to refer to ALL of these variables in, say, on the `VAR` statement in a `PROC MEANS` procedure, you can use the numbered range list `MON1-MON36`. Note the *single dash* between the first variable name and the last variable name. SAS will automatically pick up all of the consecutively numbered variables.

Name Range Lists

Not all of your variables will be numbered consecutively as above. Let's say you have the following variables in your SAS dataset: `TOTAL_AMT_PAID`, `COPAY_AMT`, `INGREDIENT_COST_PAID`, `PATIENT_PAID_AMT_SUBMITTED`, `PATIENT_PAY_AMT`, and `GROSS_AMT_DUE`. You can refer to this entire list as: `TOTAL_AMT_PAID--GROSS_AMT_DUE`. Note the *two dashes* now! The order is determined by the order in which the variables are stored in the dataset.

The Data Step – SAS Expressions and Functions

Name Range Lists (*continued*)

Note that you may have several character variables and numeric variables thrown in between these lists of variables. You can pickup only the numeric SAS variables by using this syntax: `X-numeric-Y` where X is your first numeric variable and Y is your last numeric variable. The `numeric` keyword will ensure that SAS will not throw a character variable in there. Also, if you want to pick up only the character variables, try this: `A-character-Z`.

Name Prefix Lists

To pick up all variables that begin with the same few characters, you can use the name prefix lists. For example, if your variables are `MON_JAN06, MON_FEB06, MON_MAR06, MON_APR06`, etc. , you can refer to them collectively as `MON:`. Note that the text `MON` is common among all of the variables, so I used `MON` and followed it by a colon.

Special Name Lists

There are 3 special names you can use in your programs:
`_NUMERIC_` will pick up all of your numeric variables
`_CHARACTER_` will pick up all of your character variables
`_ALL_` will pick up ALL of your variables.

Note: Some SAS functions require the `OF` keyword to work with a variable list:

```
TOT_YEAR=sum(OF MON1-MON12);
```

The Data Step – Control Structure Redux

In a previous section, we discussed some of the control structures such as IF-THEN-ELSE, DO END, and SELECT. There are a few more and we will discuss the remaining control structures in this section.

Iterative DO

Occasionally, you will need to iterate (or loop) several times to produce the expression or output you want. The syntax for the Iterative DO is:

```
DO loopvar = starting-value TO ending-value BY skip-value
   data step statements go here
END;
```

`Loopvar` can be any legal SAS variable name. We usually use "I".

`Starting-Value` is the beginning value of the loop (usually 1)

`Ending-Value` is the ending value of the loop

`BY skip-value` is an optional statement that tells the loop to use a skip pattern rather than just incrementing by 1 all the time.

Here is an example:

```
data MyTestDataset;
 do I=1 to 10;
  MyVar1=2**I;
  output;
 end;
run;
```

The Data Step – Control Structure Redux

Iterative DO (continued)

You can also specify a comma-delimited list of values instead of specifying the start and ending values:

```
data MyTestDataset;
 do I=1,3,5,7,9;
  MyVar1=2**I;
  output;
 end;
run;
```

Note that the above cheesy example is equivalent to this cheesy example:

```
data MyTestDataset;
 do I=1 to 10 BY 2;
  MyVar1=2**I;
  output;
 end;
run;
```

The Data Step – Control Structure Redux

DO WHILE

The DO WHILE statement allows you to continue to loop as long as the WHILE expression is true; otherwise, it stops looping. Contrast this with the Iterative DO which loops based on the starting and ending values. But, you might not know these values up front, so that's when you would use a DO WHILE loop. For example, instead of using the FACT(x) function to compute *x!*, let's do it the old fashioned way by computing *5!* using a DO WHILE loop:

```
data Fact5(keep=N factN);
 N=5;
 factN=1;
 iter=1;
 do while(iter <= N);
  factN=factN*iter;
  iter=iter+1;
 end;
 output;
run;
```

DO UNTIL

This is similar to DO WHILE, but will *always* perform the loop statements at least once. The expression is evaluated at the bottom of the loop, unlike the DO WHILE where it is evaluated at the top!

The Data Step – Using Arrays in the Data Step

Occasionally, you will need to perform a similar calculation on a lot of variables. For example,

```
data MyBigDataset;
 MyVar1=1;MyVar2=2;MyVar3=3;MyVar4=4;MyVar5=5;MyVar6=6;MyVar7=7;MyVar8=8;
 MyVar1=MyVar1+1;
 MyVar2=MyVar2+1;
 MyVar3=MyVar3+1;
 MyVar4=MyVar4+1;
 MyVar5=MyVar5+1;
 MyVar6=MyVar6+1;
 MyVar7=MyVar7+1;
 MyVar8=MyVar8+1;
run;
```

As you can see, we have to perform the "add one" formula 8 times, once for each variable. Just imagine if we had 100 or 500 variables! An easy way to perform these types of repetitive calculations is with SAS Arrays. An array is a "shorthand" syntax to help refer to a lot of variables without having to type in each one by hand. Note that this shorthand name is not a permanent part of the SAS dataset, it's just there temporarily to help you out. Here is an example of how to use an array to perform the calculation above:

The Data Step – Using Arrays in the Data Step


```
data MyBigDataset;
 MyVar1=1;MyVar2=2;MyVar3=3;MyVar4=4;MyVar5=5;MyVar6=6;MyVar7=7;MyVar8=8;
 array allvars{*} MyVar1-MyVar8;
 do i=1 to 8;
  allvars{i} = allvars{i} + 1;
 end;
run;
```

As you can see, this example uses both the ARRAY statement as well as an Iterative DO Loop.

Now, it might seem that the only way you can use a SAS Array is by associating pre-existing SAS variables with the array. There are two ways around this:

1. You can create a SAS Array that creates SAS Variables without referring to pre-existing SAS Variables.
2. You can create a *temporary* SAS array not associated with SAS variables and that will not create SAS Variables

Here is an example of #1:

```
array MyVars{5};
```

This will create five SAS Variables MyVar1, MyVar2, MyVar3, MyVar4 and MyVar5.

The Data Step – Using Arrays in the Data Step

Here is an example of #2:

```
array MyVars{5} _temporary_;
```

This will **NOT** create five SAS Variables MyVar1, MyVar2, MyVar3, MyVar4 and MyVar5 as in the previous example, but you can use these variables as you see fit in your SAS program to perform calculations, etc.

One easy way to initialize arrays, especially in the last two examples, is to use an *initial value list*. An initial value list is a comma-delimited list of values surrounded by parentheses that follows the array definition. For example,

```
data MyData;
 array MyVars{5} (1,2,3,4,5);
run;
```

Now, the variable MyVars1 is set to 1, …, MyVars5 is set to 5.

The Data Step – Using the SET Statement

You should all be in awe that up to now this lecture never involved reading in a pre-existing SAS dataset! That's all gonna change now!!

Once you've created one, two, or more SAS datasets, there is a high degree of probability (or a low degree of unlikeliness) that you will want to create a new SAS dataset from a pre-existing SAS dataset, either by reading in the whole pre-existing SAS dataset or some portion of it. This is where the SET statement comes in. The SAS SET Statement reads in observations from one or more SAS datasets. Although you can do more with the SET statement, we will concentrate on how to append SAS datasets.

Given the SAS datasets `MyData1`, `MyData2`, and `MyData3`, we want to create one SAS Dataset containing ALL of the data in these three SAS Datasets. Naturally, all three SAS datasets should contain the same variables, but that's not necessarily a requirement, as we shall see. For example:

```
data MyData1;
 NDC_KEY='11111111111';output;
run;

data MyData2;
 NDC_KEY='22222222222';output;
run;

data MyData3;
 NDC_KEY='33333333333';output;
run;
```

The Data Step – Using the SET Statement


```
data ALLDATA;
 set MyData1
     MyData2
     MyData3;
run;
```

Once this Data Step completes, you will have a new SAS Dataset called `ALLDATA` that contains all of the rows in `MyData1`, `MyData2` and `MyData3`. Here is what `ALLDATA` looks like:

NDC_KEY
11111111111
22222222222
33333333333

For those of you familiar with SQL, this is similar to the UNION ALL statement.

Note that we only had one variable, NDC_KEY, in the example above, but the SET statement works with as many variables as the traffic will allow.

But, what does SAS do if confronted with one or more datasets containing one or more variables that do not exist in the other datasets? SAS will happily append all of the data together, but make the data missing (i.e., NULL) where those variables did not exist in the other datasets. For example,

The Data Step – Using the SET Statement


```
data MyData1;
 NDC_KEY='11111111111';COPAY_AMT=5;
 output;
run;

data MyData2;
 NDC_KEY='22222222222';PATIENT_PAY_AMT=25;
 output;
run;

data MyData3;
 NDC_KEY='33333333333';MissPiggy='Moi, dammit!';
 output;
run;

data ALLDATA;
 set MyData1
     MyData2
     MyData3;
run;
```

NDC_KEY	COPAY_AMT	PATIENT_PAY_AMT	MissPiggy
11111111111	5	.	
22222222222	.	25	
33333333333	.	.	Moi, dammit!

The Data Step – Using the SET Statement

One additional function of the SET statement is to *interleave* data; that is, to append two (or more) datasets so that the ordering of the two (or more) datasets remains. You visualize this as the interleaving of two halves of a deck of cards by an expert card player. For example,

```
data MyData1;
 NDC_KEY='11111111111';output;
 NDC_KEY='33333333333';output;
run;

data MyData2;
 NDC_KEY='22222222222';output;
 NDC_KEY='44444444444';output;
run;

data ALLDATA;
 set MyData1
     MyData2;
 by NDC_KEY; /* Take note of the BY Statement!! */
run;
```

NDC_KEY
11111111111
22222222222
33333333333
44444444444

The Data Step – Using the SET Statement

Another useful function of the SET statement is to bring one row of data from one SAS Dataset (say, grand totals), and attach that row to ALL of the rows of another SAS Dataset For example,

```
data NDCSubTotals;
 NDC_KEY='11111111111';NDC_SubTotal=1000;output;
 NDC_KEY='22222222222';NDC_SubTotal=2000;output;
run;

data NDCTotal;
 NDC_TOTAL=3000;output;
run;

data ALLDATA;
 if _n_=1 then set NDCTotal;
 set NDCSubTotals;
run;
```

NDC_TOTAL	NDC_KEY	NDC_SubTotal
3000	11111111111	1000
3000	22222222222	2000

The Data Step – Using the SET Statement

Note that if you use the SET statement more than once in a Data Step, the LAST dataset is used to create the new dataset. This is in marked contrast to using a SET statement once with multiple SAS datasets (appending).

As you could have probably guessed, there are a lot of options to the SET statement. Here is one important one:

END=*variable-name*

This option sets *variable-name* to 1 when SAS reaches the last row in the SAS Dataset; otherwise, it is 0. Note that v*ariable-name* is temporary. For example,

```
data MyOldData;
 PATIENT_KEY=1;output;
 PATIENT_KEY=2;output;
 PATIENT_KEY=3;output;
run;

data MyNewData;
 set MyOldData end=lastcase;
 if lastcase then do;
  Total_Records=_n_;
  output;
 end;
 else delete;
run;
```

The Data Step – Using the MERGE Statement

There may be times when you want to merge together two or more SAS datasets (horizontally) instead of appending (vertically). In this case, use the SAS MERGE Statement. Just as we've seen with the SET Statement, there are times you need to have a BY Statement and times when you don't need it.

For example, let's say we have one dataset containing a list of PATIENT_KEYs and another dataset containing patient information and we want to merge the patient information with our list of PATIENT_KEYs to create one final dataset.

```
data PatientKeyList;
 PATIENT_KEY=123;output;
 PATIENT_KEY=456;output;
 PATIENT_KEY=789;output;
run;

data PatientInfo;
 PATIENT_KEY=123;PAT_GENDER="M";PAT_ZIP="12345";output;
 PATIENT_KEY=789;PAT_GENDER="F";PAT_ZIP="67890";output;
 PATIENT_KEY=901;PAT_GENDER="M";PAT_ZIP="23456";output;
run;

proc sort data=PatientKeyList;
 by PATIENT_KEY;
run;

proc sort data=PatientInfo;
 by PATIENT_KEY;
run;

data FINAL_PATS_AND_INFO;
 merge PatientKeyList PatientInfo;
 by PATIENT_KEY;
run;
```

The Data Step – Using the MERGE Statement

Here is the result of this MERGE:

PATIENT_KEY	PAT_GENDER	PAT_ZIP
123	M	12345
456		
789	F	67890
901	M	23456

Now, take note of three things:

- We now have the two fields PAT_GENDER and PAT_ZIP in our new dataset.
- Notice, though, that we have missing values (blanks) for PAT_GENDER and PAT_ZIP for PATIENT_KEY=456.
- Notice that even though PATIENT_KEY=901 does NOT appear in our list of patients, it appears in the resulting SAS dataset.

If you are familiar with SQL, you may recognize that the MERGE is behaving like a FULL OUTER join…it includes all data from both datasets.

Clearly, we cannot live like this! You can use the SAS Dataset Option IN= on both of your datasets to help control whether you want all of the data from the left dataset, the right dataset or both datasets.

The Data Step – Using the MERGE Statement

Here is that MERGE again, but this time we keep only the data matching exactly between the two datasets:

```
data FINAL_PATS_AND_INFO;
 merge PatientKeyList(in=inL) PatientInfo(in=inR);
 by PATIENT_KEY;
 if inL & inR;
run;
```

Here is the result:

PATIENT_KEY	PAT_GENDER	PAT_ZIP
123	M	12345
789	F	67890

Take note of the IN= Dataset Option in both datasets. With this option, you give a name of a (temporary) variable that will be 1 if the current row is from that dataset; otherwise, 0. Our two temporary variables are inL and inR. Take note of the Subsetting If Statement above. This tells SAS to return rows that are in both the left and right datasets. Those records that do not meet that criteria are not placed in the new dataset. (SQL Heads: Think INNER JOIN!)

Obviously, if we wanted to keep all of the data in PatientKeyList, even though there might not be a match in the PatientInfo dataset, we can change our Subsetting If Statement to read: if inL; (SQL Heads: Think LEFT JOIN!)

Note that you can merge more than two datasets together at once.

The Data Step – Using the *FIRST.* and *LAST.* Variables

Suppose you have a SAS Dataset that contains a distinct list of PATIENT_KEY and COPAY_AMT combinations. Suppose you want to know the minimum COPAY_AMT per PATIENT_KEY? How about the maximum COPAY_AMT? Suppose you want to keep at most the three lowest COPAY_AMTs per PATIENT_KEY? These questions can be programmed using the FIRST. and LAST. variables within a SAS Data Step. These variables are temporary, which means SAS creates them for the duration of the SAS Data Step and they are deleted at the end of the SAS Data Step. In order to use FIRST. and LAST. your data needs to be sorted by the relevant variable (PATIENT_KEY in our example) and you need to use a BY Statement on the relevant variable. Here is an example:

```
data PatientKeyCopayAmt;
 PATIENT_KEY=123;COPAY_AMT=10;output;
 PATIENT_KEY=123;COPAY_AMT=20;output;
 PATIENT_KEY=123;COPAY_AMT=30;output;
 PATIENT_KEY=123;COPAY_AMT=40;output;
 PATIENT_KEY=123;COPAY_AMT=50;output;

 PATIENT_KEY=234;COPAY_AMT=15;output;
 PATIENT_KEY=234;COPAY_AMT=25;output;
 PATIENT_KEY=234;COPAY_AMT=35;output;
 PATIENT_KEY=234;COPAY_AMT=45;output;
 PATIENT_KEY=234;COPAY_AMT=55;output;

run;
```

The Data Step – Using the *FIRST.* and *LAST.* Variables


```
proc sort data=PatientKeyCopayAmt;
 by PATIENT_KEY COPAY_AMT;
run;

data LowestCopayAmt;
 set PatientKeyCopayAmt;
 by PATIENT_KEY;
 if FIRST.PATIENT_KEY then output;
 else delete;
run;
```

Take note that in order to find the lowest COPAY_AMT for each PATIENT_KEY, we not only sort by the PATIENT_KEY but also by the COPAY_AMT. This will force the lowest COPAY_AMT to appear first for each patient.

Also, note that we are using FIRST.PATIENT_KEY along with an OUTPUT Statement. SAS will set FIRST.PATIENT_KEY temporary variable to a 1 for the first record of each PATIENT_KEY; otherwise, it is set to 0. Similarly for LAST.PATIENT_KEY: SAS will set LAST.PATIENT_KEY temporary variable for the last record of each PATIENT_KEY; otherwise, 0.

Obviously, in this case, FIRST.PATIENT_KEY will return the lowest COPAY_AMT for each patient, whereas LAST.PATIENT_KEY will return the highest COPAY_AMT for each patient. This is because of the way we sorted the data!

Finally, the dataset LOWESTCOPAYAMT contains ONE row per patient!

The Data Step – RETAIN and Accumulator Variables

As we just saw, we can use the FIRST. and LAST. variables to output the FIRST or LAST record for each BY variable. Suppose you have multiple records per PATIENT_KEY, along with other information such as DATE_OF_SERVICE and COPAY_AMT, and you want to calculate the *average daily copay* based on the *total* COPAY_AMT for that patient (across all DATE_OF_SERVICE) divided by the number of days between the first and last DATE_OF_SERVICE for that patient. We then want to output that *final* record for the patient with our average daily copay amount.

First, we should sort the dataset by PATIENT_KEY and DATE_OF_SERVICE. We talk more about sorting later on in the presentation.

In order to accumulate the total COPAY_AMT for the patient, we use an accumulator variable. This will add up (or accumulate) the COPAY_AMT starting at the FIRST.PATIENT_KEY record until we need the total at the LAST.PATIENT_KEY record.

In order to compute the number of days between first and last DATE_OF_SERVICE we need to know the first DATE_OF_SERVICE as well as the last DATE_OF_SERVICE. If we sort the data by PATIENT_KEY and DATE_OF_SERVICE, the last DATE_OF_SERVICE will be on the record where LAST.PATIENT_KEY=1. But! The first DATE_OF_SERVICE will be on the first record. How do we keep that first DATE_OF_SERVICE until we need to use it? We use a RETAIN Statement to tell SAS that we want to repeat that piece of information on all records starting with the FIRST.PATIENT_KEY until the LAST.PATIENT_KEY.

The Data Step – RETAIN and Accumulator Variables


```
data PatientKeyCopayAmt;

 PATIENT_KEY=123;DATE_OF_SERVICE='01JAN2006'd;COPAY_AMT=10;output;
 PATIENT_KEY=123;DATE_OF_SERVICE='01FEB2006'd;COPAY_AMT=20;output;
 PATIENT_KEY=123;DATE_OF_SERVICE='01MAR2006'd;COPAY_AMT=30;output;
 PATIENT_KEY=123;DATE_OF_SERVICE='01APR2006'd;COPAY_AMT=40;output;
 PATIENT_KEY=123;DATE_OF_SERVICE='01MAY2006'd;COPAY_AMT=50;output;

 PATIENT_KEY=234;DATE_OF_SERVICE='01JAN2007'd;COPAY_AMT=15;output;
 PATIENT_KEY=234;DATE_OF_SERVICE='01FEB2007'd;COPAY_AMT=25;output;
 PATIENT_KEY=234;DATE_OF_SERVICE='01MAR2007'd;COPAY_AMT=35;output;
 PATIENT_KEY=234;DATE_OF_SERVICE='01APR2007'd;COPAY_AMT=45;output;
 PATIENT_KEY=234;DATE_OF_SERVICE='01MAY2007'd;COPAY_AMT=55;output;

run;

proc sort data=PatientKeyCopayAmt;
 by PATIENT_KEY DATE_OF_SERVICE;
run;


…continued…
```

The Data Step – RETAIN and Accumulator Variables


```
data AverageDailyCopay;

 retain FIRST_DATE_OF_SERVICE;
 format DATE_OF_SERVICE FIRST_DATE_OF_SERVICE mmddyy10.;

 set PatientKeyCopayAmt;
 by PATIENT_KEY;

 /* Accumulate the COPAY_AMT for each PATIENT_KEY */
 if FIRST.PATIENT_KEY then do;
  TotalCopayAmt=0;
 end;

 TotalCopayAmt + COPAY_AMT; /* TotalCopayAmt initialized to zero!! */

 /* Let`s retain the first DATE_OF_SERVICE */
 if FIRST.PATIENT_KEY then do;
  FIRST_DATE_OF_SERVICE=DATE_OF_SERVICE;
 end;

 /* Perform the Average Copay Amount */
 if LAST.PATIENT_KEY then do;
  AverageCopayAmt=TotalCopayAmt/(DATE_OF_SERVICE-FIRST_DATE_OF_SERVICE);
  output;
 end;

run;
```

The Data Step – CONTINUE and LEAVE Statements

There are two additional SAS statements which give you more control over Do-Loops: CONTINUE and LEAVE.

The CONTINUE statement, from within a Do-Loop, tells SAS to stop the current iteration of the Do-Loop, go back to the top of the Do-Loop and perform the next loop.

The LEAVE statement tells SAS to stop the Do-Loop completely and continue with the next statement appearing after the Do-Loop.

Example (CONTINUE)

```
data test1;
 do i=1 to 100;
  if i=50 then continue;
  else do;
   z=i;
   output;
  end;
 end;
run;
/* 99 observations */
```

Example (LEAVE)

```
data test2;
 do i=1 to 100;
  if i=50 then leave;
  else do;
   z=i;
   output;
  end;
 end;
run;
/* 49 Observations */
```

The Data Step – Using DROP=, KEEP=, RENAME=

There are three additional important SAS dataset options you should be aware of. The DROP=, KEEP= and RENAME= statements control which variables to *keep* in a dataset once SAS has completed building it, which variables to *drop* from a dataset once SAS has completed building it, and which variables to *rename*.

Note that you can use these statements on **both** incoming and outgoing datasets!

Note that there are corresponding Data Step Statements DROP, KEEP and RENAME (no equal signs!), but they refer to the outgoing dataset ONLY which is why we prefer the three dataset options instead!

```
data test1(keep=z drop=i rename=(z=Bob_The_Variable));
 do i=1 to 100;
  if i=50 then continue;
  else do;
   z=i;
   output;
  end;
 end;
run
```

Note the syntax for the RENAME= Statement:

```
RENAME=(oldvar=newvar …)
```

The Data Step – How SAS Handles Missing Values

Not all of your data is going to be perfect and many variables will contain, what SAS calls, *missing values*. Missing values are analogous to NULLs in the database world; that is, variables with values that are unknown.

SAS represents missing numeric values by a period "." when you view a dataset or print a report. SAS represents a character value by a blank space.

Unlike SQL Server and Oracle, SAS has the ability to represent different *categories* of missing values. These are called "Special Missing Values" and are represented by .A, .B, …, .Z and ._ (underscore). You can use these special missing values in an assignment statement.

If the data you are reading in contains, (say) an "M" (for missing), and (say) "I" (for incomplete), or (say) "U" (for unknown), you can tell SAS ahead of time that these three letters represent missing values by using the MISSING statement in your Data Step.

When using SAS Functions, missing values are taken into account automatically. So, for example, if you want to find the mean across twelve months of data, but one or more variable values is missing, SAS will take that into account and change the calculation. So, AvgDollars=MEAN(OF JAN-DEC) will compute the average dollars from January to December, but if JAN is missing, the denominator is adjusted to 11 instead of 12.

Note that missing values appearing in calculations involving arithmetic operators *results in a missing value*. So, for TOT=A+5, if A is missing, TOT is missing. But, TOT=SUM(A,5) is 5 and not missing.

The Data Step – How SAS Handles Missing Values

Note that missing values and special missing values get their own sort order and is (from smallest to largest): ._, ., .A, .B, …, .Z, negative numbers follow, zero follows, positive numbers follow.

For character variables, the missing value (a blank) is sorted before alphanumeric values.

Naturally, SAS will set a variable's value to missing if there is a division by zero, an illegal value in a function, illegal character to numeric conversion, etc.

You can set a numeric variable to missing yourself by placing a period after the equal sign: `if Gender="U" then TotDollars=.;`

You can set a character variable to missing yourself by placing a " " after the equal sign: `if Gender="U" then State=" ";`

You can check for "missing-ness" in logical expressions by comparing to a period or a blank:

```
if TotDollars=. then…
If Gender=" " then …
```

You can also use the MISSING() function to check if a variable is missing:

```
if missing(myvar) then /* missing */
```

Missing values are considered FALSE in logical expressions. So, `if (BOB AND WILMA) then…` will be false if either BOB or WILMA or both are missing.

The Data Step – How SAS Handles Missing Values


```
data CompanySalary;
 EmployeeStatus="DEAD";Salary=500;output;
 EmployeeStatus=" ";Salary=100;output;
 EmployeeStatus="ACTIVE";Salary=200;output;
 EmployeeStatus="ACTIVE";Salary=300;output;
run;

data TotalDollars;
 set CompanySalary end=lastcase;
 if EmployeeStatus="DEAD" then Salary=.;
 else if EmployeeStatus=" " then Salary=.;
 TotSal+Salary;
 if lastcase then do;
  putlog "TotSal=" TotSal;
 end;
run;
```

The Data Step – Using the WHERE= Dataset Option

When you need to subset your data, instead of using an IF THEN ELSE statement, which is slow, you can use the WHERE= dataset option which is supposed to be faster. Here is an example:

```
data CompanySalary;
 EmployeeStatus="DEAD";Salary=500;output;
 EmployeeStatus=" ";Salary=100;output;
 EmployeeStatus="ACTIVE";Salary=200;output;
 EmployeeStatus="ACTIVE";Salary=300;output;
run;

data DEAD_EMPLOYEES;
 set CompanySalary(where=(EmployeeStatus="DEAD"));
run;
```

Working with SAS Dates and Times

SAS Dates and Times confuse people a lot! This may be caused by the fact that, unlike Oracle and SQL Server, there is no DATE data type in SAS, but only numbers and characters.

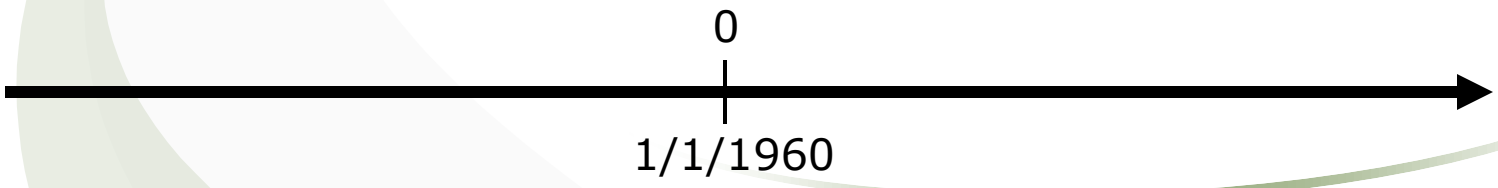
So, how does SAS represent dates and times?

SAS represents dates (without a time portion) as the ***number of days*** since January 1, 1960.

SAS represents date/times as the ***number of seconds*** since January 1, 1960.

Again, since there is no native DATE/DATETIME data type in SAS, and since SAS uses **numbers** to represent dates and date/times, *it's up to the user to keep track of which variables represent dates and which variables that represent date/times in your SAS datasets.*

In this section, we'll discuss how to create SAS date and date/time variables, how to perform calculations on dates and date/times, introduce some SAS functions you can use with dates and date/times, and how to work with SAS dates and date/times when pulling date and date/time data from a database table, and pushing date and date/time data to a database table.



Working with SAS Dates and Times

First, let's see how to create a SAS date variable from a particular date. Below, I use the `INPUT()` function and the `mmddyy10.` SAS Informat to read in a date and create a SAS Date value.

```
data date1;
 sdv1=input("01/02/1960",mmddyy10.); /* SAS Date Value:     1 */
 sdv2=input("12/31/1990",mmddyy10.); /* SAS Date Value: 11322 */
run;
```

Note that the value `11322` represents the number of days from `1/1/1960` to `12/31/1990`.

As you might have guessed, there are several SAS Informats you can use to read in dates:

- `MMDDYYw.` = Expects the date to look like *mmddyy* or *mmddyyyy*
- `DDMMYYw.` = Expects the date to look like *ddmmyy* or *ddmmyyyy*
- `YYMMDDw.` = Expects the date to look like *yymmdd* or *yyyymmdd*
- `YYMMNw.` = Expects the date to look like *yymm* or *yyyymm*
- `DATEw.` = Expects the date to look like *ddmonyy* or *ddmonyyyy*

Note that some of these informats can handle slashes, dashes, etc. in the date.

```
data date1;
 sdv1=input("01/02/1960",mmddyy10.); /* 11322 */
 sdv2=input("12/31/1990",mmddyy10.); /* 11322 */
 sdv3=input("31121990",ddmmyy8.); /* 11322 */
 sdv4=input("19901231",yymmdd8.); /* 11322 */
 sdv5=input("199012",yymmn6.); /* 11292 (first day of month!) */
 sdv6=input("31DEC1990",date9.); /* 11322 */
run;
```

Working with SAS Dates and Times

Another way to create a SAS Date value is to use the function `MDY()`. This function expects three parameters: the month number of the year, the day number of the year, and the year number.

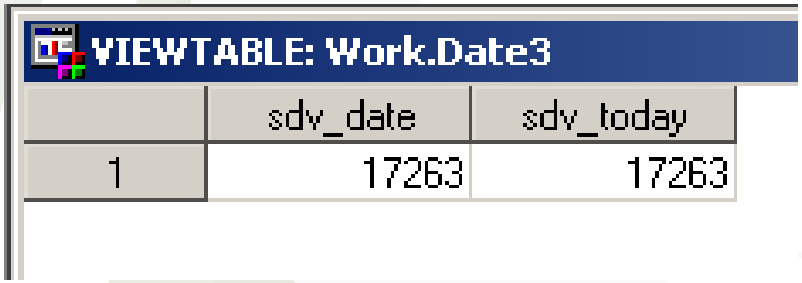
```
data date1;
 sdv=mdy(12,31,1990); /* SAS Date Value: 11322 */
run;
```

For those of you who are curious, you can get the current date by using the `DATE()` or `TODAY()` functions:

```
data date3;
 sdv_date=date(); /* 17263 */
 sdv_today=today(); /* 17263 */
run;
```

Note that both functions do not expect any parameters.

So far we've learned how to create a SAS Date Value from a character string using informats and SAS functions. Unfortunately, when you open up your SAS dataset, you will see the following:



VIEWTABLE: Work.Date3

	sdv_date	sdv_today
1	17263	17263

As you can see, the SAS Date Values appear and not a readable text version of the SAS Date Value. This is where using a SAS Format comes in. A SAS Format allows you to "see" the SAS Date Value in a readable form.

Working with SAS Dates and Times

There are several SAS Formats you can use on SAS Date Values. Here is an example of one of them:

```
data date3;
 format sdv_date sdv_today mmddyy10.;
 sdv_date=date(); /* 17263 */
 sdv_today=today(); /* 17263 */
run;
```

VIEWTABLE: Work.Date3

	sdv_date	sdv_today
1	04/07/2007	04/07/2007

Applying the format `MMDDYY10.` to both `SDV_DATE` and `SDV_TODAY` during the creation of the SAS dataset allows you to see the SAS Date Value in a more familiar form.

Naturally, there are several other SAS formats you can use to display SAS Date Values in more pleasing forms:

- `DATEw.` = Displays SAS Date Values like *ddmonyy* or *ddmonyyyy*
- `DDMMYYw.` = Displays SAS Date Values like *ddmmyy* or *ddmmyyyy*
- `MMDDYYw.` = Displays SAS Date Values like *mmddyy* or *mmddyyyy*
- `YYMMDDw.` = Displays SAS Date Values like *yymmdd* or *yyyy*mmdd
- `MMDDYY`**`x`**`w.` = Displays SAS Date Values like *mmddyy* or *mmddyyyy or mmBddByyyy depending on x (B=blank, C=colon, D=dash, N=no separator, P=period, S=slash)*

```
data date3;
 format sdv_date sdv_today mmddyyD10.;
 sdv_date=date(); /* 17263 displays as 04-07-2007 */
 sdv_today=today(); /* 17263 displays as 04-07-2007 */
run;
```

VIEWTABLE: Work.Date3

	sdv_date	sdv_today
1	04-07-2007	04-07-2007

Working with SAS Dates and Times

So far, we've created SAS Date Values using the `INPUT()` function and SAS Informats, used the `MDY()` function, and used SAS Formats to aid in the display of the SAS Date Values when you open up a SAS dataset to peruse it.

One task we haven't talked about yet is to create a character variable containing a formatted version of a SAS Date Value. You may want to do this for several reasons, one of which is with Oracle/SQL Server dates in pass-through SQL.

In order to create a character variable from a SAS Date Value, you use the SAS `PUT()` function along with the SAS Formats we mentioned on the previous slide. Don't forget to set the *length* of your SAS Character variable appropriately. For example,

```
data date3;
 length strDate strToday $ 10 strDate2 $ 9;
 sdv_date=date(); /* 17263 */
 sdv_today=today(); /* 17263 */
 strDate=put(sdv_date,mmddyy10.); /* 04/07/2007 */
 strToday=put(sdv_today,mmddyyD10.); /* 04-07-2007 */
 strDate2=put(sdv_date,date9.); /* 07APR2007 */
run;
```

VIEWTABLE: Work.Date3

	strDate	strToday	strDate2	sdv_date	sdv_today
1	04/07/2007	04-07-2007	07APR2007	17263	17263

Working with SAS Dates and Times

One thing you should keep in mind is that most of the SAS Date Values you will work with on a day-to-day basis are in the *tens of thousands* – recall that `4/7/2007` was represented as the SAS Date Value `17263` in the previous examples – whereas SAS Datetime Values are in the *millions or billions*. For example, the SAS Datetime Value for midnight `4/7/2007` is `1,491,523,200`. When working with SAS Dates and SAS Datetime Values, remember these facts and you'll know whether you've made a mistake somewhere.

We can approximate the SAS Date Value for `17263` by multiplying the number of years since `1/1/1960` by `365` days/year and then add the 4 months: `47*365+3*30+7=17,252`. Naturally, this ignores leap years and the differing number of days per month, but is an excellent way to check your work. Similar for the SAS Datetime Value: `(47*365+3*30+7)*24*60*60=1,490,590,052`.

Next, we will talk about SAS Datetime Values. One thing you'll notice about this section is how similar it is to the previous section on SAS Date Values. Sure, the formats, informats and functions may be slightly different, but the ideas are exactly the same. Recall that a SAS Datetime Value is the *number of seconds* since `1/1/1960` whereas a SAS Date Value is the *number of days* since `1/1/1960`.

Let's assume we have a SAS Date Value and we want to convert it to a SAS Datetime Value. This type of conversion is VERY useful when you want to load data into an Oracle or SQL Server database table where one or more of the columns are DATE/DATETIME data types. Here's an example where we use the `DHMS()` function:

```
data date4;
 sdtv1=dhms(date(),0,0,0); /* SAS Datetime Value: 1491523200 (0 hours,0 minutes,0 seconds) */
 sdtv2=dhms(input("04/07/2007",mmddyy10.),0,0,0); /* SAS Datetime Value: 1491523200 */
run;
```

Working with SAS Dates and Times

Now, similar to the INPUT() function and SAS Informats we used for SAS Date Values, we can do something similar to SAS Datetime Values:

```
data date5;
 sdtv1=input("07APR2007:00:00:00",datetime18.); /* 1491523200 */
run;
```

Note that you have to specify the time portion as HH:MM:SS after the date. The `DATETIMEw.` format expects `ddmonyyyy:hh:mm:ss`.

Once again, if we go to open up the SAS dataset, we see the following:

VIEWTABLE: Work.Date5

	sdtv1
1	1491523200

Similar for SAS Date Values, we can specify a SAS Format to be used to display the SAS Datetime Value in a more readable way:

```
data date5;
 format sdtv1 datetime18.;
 sdtv1=input("07APR2007:00:00:00",datetime18.); /* 1491523200 */
run;
```

VIEWTABLE: Work.Date5

	sdtv1
1	07APR07:00:00:00

Working with SAS Dates and Times

Another useful SAS Datetime format is the `DATEAMPMw`. Format which displays the datetime with an AM or PM indicator:

```
data date5;
 format sdtv1 dateampm.;
 sdtv1=input("07APR2007:00:00:00",datetime18.); /* 1491523200 */
run;
```

VIEWTABLE: Work.Date5

	sdtv1
1	07APR07:12:00:00 AM

Naturally, you can use the PUT() function with the appropriate SAS Formats to create a character string containing your formatted SAS Datetime Value:

```
data date5;
 format sdtv1 dateampm.;
 length DateTimeString $ 19;
 sdtv1=input("07APR2007:00:00:00",datetime18.); /* 1491523200 */
 DateTimeString=put(sdtv1,dateampm19.);
run;
```

VIEWTABLE: Work.Date5

	sdtv1	DateTimeString
1	07APR07:12:00:00 AM	07APR07:12:00:00 AM

Working with SAS Dates and Times

There are some additional useful functions that might come in handy, so we will introduce them now.

Assuming that you have a variable containing SAS Date Values – these functions work **ONLY** with SAS Date Values and **NOT** Datetime Values – and you want to extract the month (as a number), the day (as a number), or the year (as a number), you can use the following functions:

- `DAY(sdv)` = returns a numeric value representing the day of the SAS Date Value
- `MONTH(sdv)` = returns a numeric value representing the month
- `YEAR(sdv)` = returns a numeric value representing the year

For example,

```
data date6;
 sdv_today=today(); /* 04/07/2007 = 17263 */
 The_Month=month(sdv_today); /* 4 */
 The_Day=day(sdv_today); /* 7 */
 The_Year=year(sdv_today); /* 2007 */
run;
```

So, how can you use these functions with SAS Datetime Values? Recall that we converted a SAS Date Value to a SAS Datetime Value using the `DHMS()` function. We can extract the SAS Date Value from a SAS Datetime Value by using the `DATEPART(sdtv)` function (you should all be familiar with this function!).

```
data date6;
 sdtv_today=dhms(today(),0,0,0); /* 1491523200 */
 The_Month=month(datepart(sdtv_today)); /* 4 */
 The_Day=day(datepart(sdtv_today)); /* 7 */
 The_Year=year(datepart(sdtv_today)); /* 2007 */
run;
```

Working with SAS Dates and Times

You can also use the `JULDATE(sdv)` function to return your date in Julian format: *yyyyddd*. The `JULDATE7(sdv)` function is the same as `JULDATE(sdv)` except it always returns a 4 digit year (and for that it is preferred).

```
data date6;
 sdv_today=today(); /* 17263 */
 Hark_Who_Goes_There=juldate7(sdv_today); /* 2007097 */
run;
```

The next thing we should talk about is how to do arithmetic with SAS Dates and Datetime Values. As we've mentioned *ad nauseum*, a SAS Date Value is the number of days since 1/1/1960 whereas a SAS Datetime Value is the number of seconds since 1/1/1960. This means, if you want to add, say one week, to a SAS Date Value, you can just add 7 to the SAS Date Value. If you want to add, say 5 minutes, to a SAS Datetime Value, you can just add 5*60=300 seconds to the SAS Datetime Value. For example,

```
data date7;
 sdv_today=today(); /* 17263 */
 sdtv_today=dhms(today(),0,0,0); /* 1491523200 */
 sdv_today_Next_Week=sdv_today+7; /* 17270 */
 sdtv_today_Five_Minutes_From_Now=sdtv_today+5*60; /* 1491523500 */
run;
```

Working with SAS Dates and Times

Of course, we can also determine the difference between two SAS Date Values and the difference between two SAS Datetime Values.

```
data date7;
 sdv_today=today(); /* 17263 */
 sdtv_today=dhms(today(),0,0,0); /* 1491523200 */
 sdv_today_Next_Week=sdv_today+7; /* 17270 */
 sdtv_today_Five_Minutes_From_Now=sdtv_today+5*60; /* 1491523500 */
 Days_Diff=sdv_today_Next_Week - sdv_today; /* 7 */
 Seconds_Diff=sdtv_today_Five_Minutes_From_Now - sdtv_today; /* 300 */
run;
```

It won't surprise you to know that there two SAS functions which allow you to shift SAS Date Values and SAS Datetime Values, as well as compute the differences between them.

The SAS function `INTNX(interval,sdv|sdtv,increment_amount)` shifts a SAS Date Value (`sdv`) or SAS Datetime Value (`sdtv`) forward `increment_amount` of `interval`s. The `interval` is a character string such as "day", "month", "year", etc.

The SAS function `INTCK(interval,from,to)` computes the number of intervals between the starting SAS Date Value/SAS Datetime Value `from` and the ending SAS Date Value/SAS Datetime Value `to`. The `interval` is a character string such as "day", "month", "year", etc.

Working with SAS Dates and Times


```
data date7;
 sdv_today=today(); /* 17263 */
 sdtv_today=dhms(today(),0,0,0); /* 1491523200 */

 sdv_today_Next_Week=sdv_today+7; /* 17270 */
 sdtv_today_Five_Minutes_From_Now=sdtv_today+5*60; /* 1491523500 */

 Days_Diff=sdv_today_Next_Week - sdv_today; /* 7 */
 Seconds_Diff=sdtv_today_Five_Minutes_From_Now - sdtv_today; /* 300 */

 sdv_today_Next_Week2=intnx('day',sdv_today,7); /* 17270 */
 Days_Diff2=intck('day',sdv_today,sdv_today_Next_Week); /* 7 */

run;
```

Note: There is much more on these two functions in the SAS documentation!

The PROC Step

The PRINT Procedure

Believe it or not, the PRINT procedure prints out the data from a SAS dataset. But, it does more than just dump the dataset, you can give it titles, footers, produce grand totals, produce sub-totals, create page breaks by certain variables, etc. Here is a basic example along with the input data:

```
proc print data=saldata width=uniform label split="*";
 var Instr Name Ysalary;
 format Ysalary dollar12.0;
 label Instr="Instrument"
       Name="Player"
       Ysalary="Yearly*Salary";
 title1 "Indonesian Symphony Orchestra";
 title2 "2005 End-of-Year Salary Disbursal by Instrument and Player";
run;
```

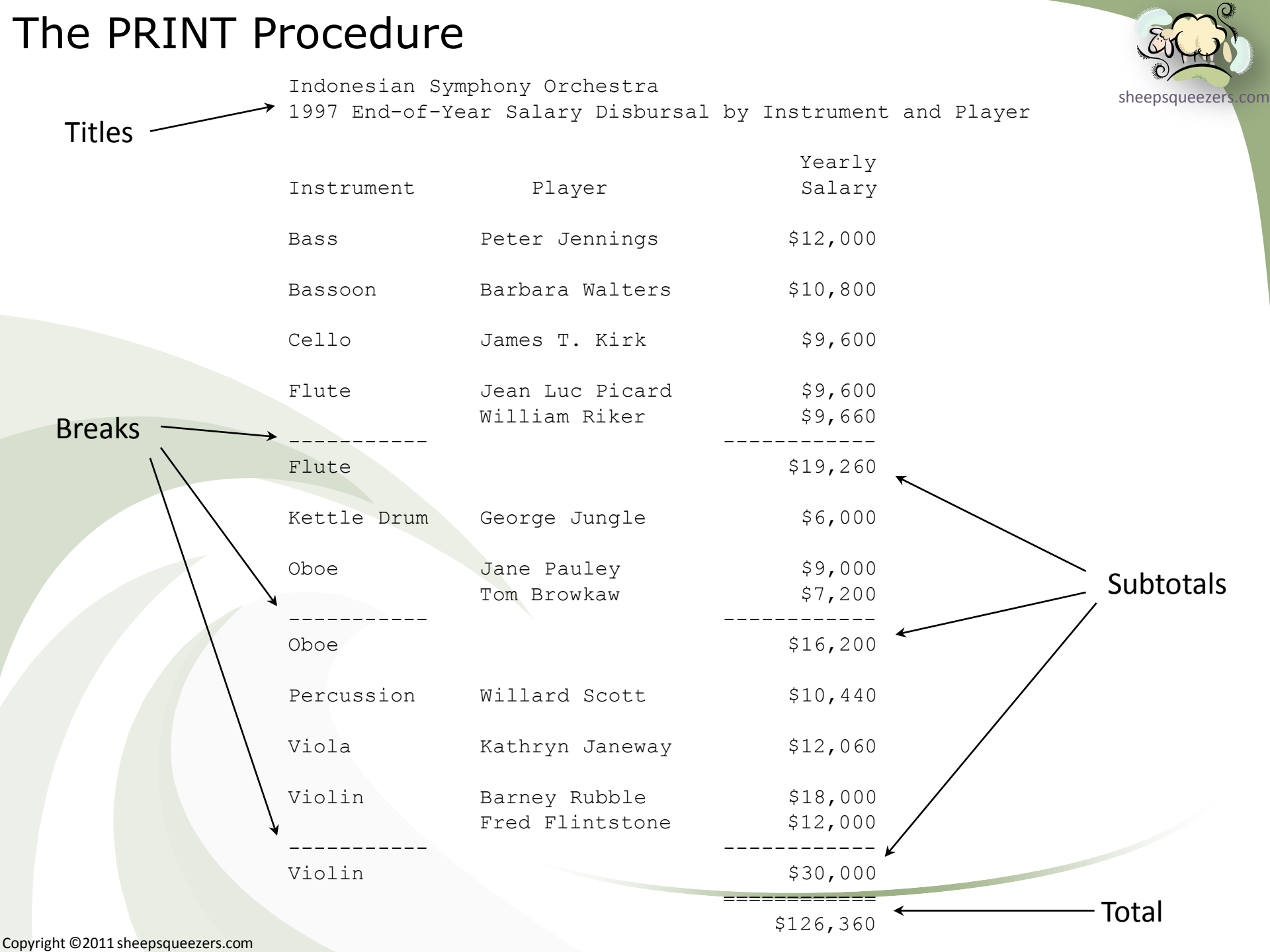
Fred Flintstone	Violin	1000
Barney Rubble	Violin	1500
George Jungle	Kettle Drum	500
Peter Jennings	Bass	1000
Tom Browkaw	Oboe	600
Barbara Walters	Bassoon	900
Jane Pauley	Oboe	750
Willard Scott	Percussion	870
James T. Kirk	Cello	800
Kathryn Janeway	Viola	1005
Jean Luc Picard	Flute	800
William Riker	Flute	805

The PRINT Procedure


```
Indonesian Symphony Orchestra
2005 End-of-Year Salary Disbursal by Instrument and Player

                                                  Yearly
Obs    Instrument          Player                 Salary

  1    Violin          Fred Flintstone          $12,000
  2    Violin          Barney Rubble            $18,000
  3    Kettle Drum     George Jungle             $6,000
  4    Bass            Peter Jennings           $12,000
  5    Oboe            Tom Browkaw               $7,200
  6    Bassoon         Barbara Walters          $10,800
  7    Oboe            Jane Pauley               $9,000
  8    Percussion      Willard Scott            $10,440
  9    Cello           James T. Kirk             $9,600
 10    Viola           Kathryn Janeway          $12,060
 11    Flute           Jean Luc Picard           $9,600
 12    Flute           William Riker             $9,660
```

```
options byline;
run;

proc print data=saldata width=uniform label split="*";
 by Instr;
 id Instr;
 var Instr Name Ysalary;
 format Ysalary dollar12.0;
 label Instr="Instrument"
       Name="Player"
       Ysalary="Yearly*Salary";
 title1 "Indonesian Symphony Orchestra";
 title2 "2005 End-of-Year Salary Disbursal by Instrument and Player";
run;
```

The PRINT Procedure



Indonesian Symphony Orchestra
1997 End-of-Year Salary Disbursal by Instrument and Player

Instrument	Player	Yearly Salary
Bass	Peter Jennings	$12,000
Bassoon	Barbara Walters	$10,800
Cello	James T. Kirk	$9,600
Flute	Jean Luc Picard	$9,600
	William Riker	$9,660
----------		----------
Flute		$19,260
Kettle Drum	George Jungle	$6,000
Oboe	Jane Pauley	$9,000
	Tom Browkaw	$7,200
----------		----------
Oboe		$16,200
Percussion	Willard Scott	$10,440
Viola	Kathryn Janeway	$12,060
Violin	Barney Rubble	$18,000
	Fred Flintstone	$12,000
----------		----------
Violin		$30,000
		==========
		$126,360

The SORT Procedure

You use the SORT procedure to sort SAS datasets by one or more variables. You can either let SAS overwrite the original dataset with the sorted version, or you can create a new dataset after the original data has been sorted.

SAS also allows you to remove duplicate rows of data based either on the sort variables only, or based on the entire record. You can choose to ignore these duplicate rows of data or have SAS write them to a SAS dataset.

```
proc sort data=MyUnSortedData out=MySortedData;
 by Var1 Var2;
run;

proc sort data=MyUnSortedData out=MySortedData NODUPKEY;
 by Var1 descending Var2;
run;

proc sort data=MyUnSortedData out=MySortedData dupout=MyDuplicateData NODUPKEY;
 by Var1 Var2;
run;
```

If you want to de-duplicate your SAS dataset based on all of the variables in that dataset rather than just the BY variable list, use the NODUPRECS option.

```
proc sort data=MyUnSortedData out=MySortedData NODUPRECS;
 by Var1 Var2;
run;
```

The SORT Procedure

Note that many people use PROC SORT NODUPKEY to obtain the first record of data in a SAS dataset. For example, let's assume we have the following data in a SAS dataset:

```
data PatientSVCData;
 format SVC_DATE mmddyy10.;
 infile datalines;
 input @1 PATIENT_KEY          4.0
       @6 SVC_DATE     MMDDYY10.;
datalines;
0001 05/01/2007
0001 01/01/2007
0002 05/01/2007
0002 04/01/2007
0002 08/01/2007
0002 10/01/2007
;
run;
```

Let's get the earliest SVC_DATE for each patient. The *correct* way to do this is:

```
proc sort data=PatientSVCData out=PatientSVCData_SORTED;
 by PATIENT_KEY SVC_DATE;
run;

data FIRST_SVC;
 set PatientSVCData_SORTED;
 by PATIENT_KEY;
 if FIRST.PATIENT_KEY;
run;
```

The SORT Procedure

Sometimes you will see SAS programmers do this:

```
proc sort data=PatientSVCData out=PatientSVCData_SORTED;
 by PATIENT_KEY SVC_DATE;
run;

proc sort data=PatientSVCData_SORTED out=PatientSVCData_SORTED_FIRST_SVC nodupkey;
 by PATIENT_KEY;
run;
```

The first SORT will sort the data by SVC_DATE within PATIENT_KEY. The second SORT will de-duplicate the data by PATIENT_KEY (notice the NODUPKEY option). This works but it depends on whether the EQUALS (maintains relative order within BY groups in the dataset) or NOEQUALS (does NOT maintain relative order within BY groups in the dataset) PROC SORT option is in effect. If you added the NOEQUALS option on the second sort along with the NODUPKEY, you would not (necessarily) get the first service record for each patient!

In conclusion, use the FIRST. concept in a Data Step!

The FREQ Procedure

The FREQ procedure produces frequency counts for variables either individually or in combination. This procedure allows you to output the frequencies into a SAS Dataset for use later in your program.

Assuming we have the following data:

PLAYER	INSTRUMENT	MSALARY
Fred Flintstone	Violin	1000
Barney Rubble	Violin	1500
George Jungle	Kettle Drum	500
Peter Jennings	Bass	1000
Tom Browkaw	Oboe	600
Barbara Walters	Bassoon	900
Jane Pauley	Oboe	750
Willard Scott	Percussion	870
James T. Kirk	Cello	800
Kathryn Janeway	Viola	1005
Jean Luc Picard	Flute	800
William Riker	Flute	805

Since this data is at the musician by instrument level, to determine the number of people who play each instrument, you code this:

```
proc freq data=SALDATA;
 table INSTRUMENT;
run;
```

The FREQ Procedure

Here are the results of the PROC FREQ:

The FREQ Procedure

INSTRUMENT	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Bass	1	8.33	1	8.33
Bassoon	1	8.33	2	16.67
Cello	1	8.33	3	25.00
Flute	2	16.67	5	41.67
Kettle Drum	1	8.33	6	50.00
Oboe	2	16.67	8	66.67
Percussion	1	8.33	9	75.00
Viola	1	8.33	10	83.33
Violin	2	16.67	12	100.00

- The "Frequency" column tells you the number of times the INSTRUMENT occurs in the dataset. For example, Flute, Oboe and Violin occur twice in the data meaning that there are two players for each one of those instruments.
- The "Percent" column shows the percentage that this INSTRUMENT's frequency count is in the data. The formula is 100*Frequency/12.
- The "Cumulative Frequency" column adds the values in Frequency column from the current INSTRUMENT on up. For example, the total number of PLAYERS for the Bass, Bassoon, Cello and Flute is 5. The last row is the sum of the Frequency column. That is, 12 is the sum of 1+1+…+1+2.
- The "Cumulative Percent" column sums up the Percent column.

The FREQ Procedure

If you would like to send the results of a PROC FREQ to a dataset, you can use this code:

```
proc freq data=SALDATA;
 table INSTRUMENT/out=FrqData noprint;
run;
```

If you have multiple variables and would like to obtain frequencies of these variables in combination, you can use this syntax:

```
proc freq data=SALDATA;
 table PLAYER*INSTRUMENT/out=FrqData noprint;
run;
```

There's a lot more to this procedure. Please see the SAS Procedures Guide for more information.

The MEANS Procedure

The MEANS procedure allows you to summarize one or more variables in your SAS dataset in a similar fashion as a GROUP BY in SQL. PROC MEANS allows you to compute the sum, min, max, mean, count, median, first quartile, third quartile, etc.

PROC MEANS also allows you to create an output SAS dataset containing the summarized data.

PROC MEANS can compute grand totals across each variable. That is, if you have one variable, you can choose to have PROC MEANS return an additional row containing the grand total along with your summarized data for that variable. If you have two or more variables, you can choose to have PROC MEANS return summarized data for each one-way, two-way, etc. combinations of variables. In order to distinguish between different combinations, SAS creates an additional variable in the out-going dataset named _TYPE_. We'll explain this variable a little later.

Assuming our data is the musician data from above, let's find the minimum, maximum and total monthly salary by instrument. You can do this using this code:

```
proc means data=SALDATA sum min max nway;
 class INSTRUMENT;
 var MSALARY;
run;
```

The MEANS Procedure

You will see the following results in the SAS Listing file:

The MEANS Procedure

Analysis Variable : MSALARY

INSTRUMENT	N Obs	Sum	Minimum	Maximum
Bass	1	1000.00	1000.00	1000.00
Bassoon	1	900.0000000	900.0000000	900.0000000
Cello	1	800.0000000	800.0000000	800.0000000
Flute	2	1605.00	800.0000000	805.0000000
Kettle Drum	1	500.0000000	500.0000000	500.0000000
Oboe	2	1350.00	600.0000000	750.0000000
Percussion	1	870.0000000	870.0000000	870.0000000
Viola	1	1005.00	1005.00	1005.00
Violin	2	2500.00	1000.00	1500.00

The MEANS Procedure

Naturally, you'll eventually want to output this data to a SAS Dataset for use later in your program. To do that, you use the OUTPUT OUT= statement like this:

```
proc means data=SALDATA nway noprint;
 class INSTRUMENT;
 var MSALARY;
 output out=INSTR_TOT sum(MSALARY)=TOT_SALARY
                      min(MSALARY)=MIN_SALARY
                      max(MSALARY)=MAX_SALARY;

run;
```

If you print out the SAS Dataset INSTR_TOT, here is what you will see:

INSTRUMENT	_TYPE_	_FREQ_	TOT_ SALARY	MIN_ SALARY	MAX_ SALARY
Bass	1	1	1000	1000	1000
Bassoon	1	1	900	900	900
Cello	1	1	800	800	800
Flute	1	2	1605	800	805
Kettle Drum	1	1	500	500	500
Oboe	1	2	1350	600	750
Percussion	1	1	870	870	870
Viola	1	1	1005	1005	1005
Violin	1	2	2500	1000	1500

Take note of the two additional variables: _TYPE_ and _FREQ_.
The variable _FREQ_ tells you the number of records that went into summarizing that particular row of data. For example, two rows from the original data SALDATA went into summarizing the VIOLIN row in the output SAS Dataset from the MEANS procedure. If you do not need this variable, you can drop it in the outgoing dataset.

The MEANS Procedure

As another example, let's assume we have this data:

```
data CITYSTATEPOPCNT;
 infile cards;
 input @1  CITY      $17.
       @18 STATE     $14.
       @32 POPCNT      4.0;
datalines;
Little Rock      Arkansas      1000
Trenton          New Jersey    2000
Princeton        New Jersey    3000
Sacramento       California    4000
Los Angeles      California    5000
Philadelphia     Pennsylvania  6000
Erie             Pennsylvania  7000
Orlando          Florida       8000
New York         New York      9000
Manhattan        New York      1000
Flushing         New York      2000
Coney Island     New York      3000
;
run;
```

And we use the following PROC MEANS code to summarize our data:

```
proc means data=CITYSTATEPOPCNT noprint;
 class CITY STATE;
 var POPCNT;
 output out=TOTPOP sum(POPCNT)=TOT_POPCNT;
run;
```

The MEANS Procedure

Note that we did not use the NWAY option. The resulting SAS dataset looks like this (except for the colors, I put those in to show the rows of data for each _TYPE_):

CITY	STATE	_TYPE_	_FREQ_	TOT_POPCNT
		0	12	51000
	Arkansas	1	1	1000
	California	1	2	9000
	Florida	1	1	8000
	New Jersey	1	2	5000
	New York	1	4	15000
	Pennsylvania	1	2	13000
Coney Island		2	1	3000
Erie		2	1	7000
Flushing		2	1	2000
Little Rock		2	1	1000
Los Angeles		2	1	5000
Manhattan		2	1	1000
New York		2	1	9000
Orlando		2	1	8000
Philadelphia		2	1	6000
Princeton		2	1	3000
Sacramento		2	1	4000
Trenton		2	1	2000
Coney Island	New York	3	1	3000
Erie	Pennsylvania	3	1	7000
Flushing	New York	3	1	2000
Little Rock	Arkansas	3	1	1000
Los Angeles	California	3	1	5000
Manhattan	New York	3	1	1000
New York	New York	3	1	9000
Orlando	Florida	3	1	8000
Philadelphia	Pennsylvania	3	1	6000
Princeton	New Jersey	3	1	3000
Sacramento	California	3	1	4000
Trenton	New Jersey	3	1	2000

The MEANS Procedure

Things become much clearer if you think of _TYPE_ in binary:

TYPE=0	In Binary➔	00
TYPE=1	In Binary➔	01
TYPE=2	In Binary➔	10
TYPE=3	In Binary➔	11

or

TYPE=0	In Binary➔	000
TYPE=1	In Binary➔	001
TYPE=2	In Binary➔	010
TYPE=3	In Binary➔	011
TYPE=4	In Binary➔	100
TYPE=5	In Binary➔	101
TYPE=6	In Binary➔	110
TYPE=7	In Binary➔	111

Notice that the positions of the 1's correspond with the position of the variables listed on the CLASS statement.

There's a lot more to this procedure. Please see the SAS Procedures Guide for more information.

The FORMAT Procedure

The FORMAT procedure allows you to create your own formats and informats.

Recall that formats and informats are used to map, say, a SAS Date Value to the text in MM/DD/YYYY format (using the `MMDDYY10.` SAS Format) or read in, say, a text value using the `COMMA12.` SAS informat.

You can either create formats "by hand" or by using an existing SAS Dataset. For example, if we want to map "M" to "Male", "F" to "Female" and all others to "Unknown", since there are only three items, we can create the format by hand:

```
proc format;
 value $m2d "M"  = "Male   "
            "F"  = "Female "
            other= "Unknown";
run;
```

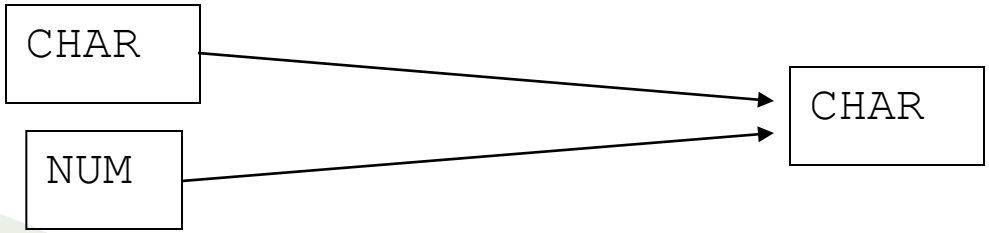
But, if your intention is to map each one of your 10,000 NDC_KEYs to its associated Label Name (LN), then doing this by hand is *probably* not a good idea. But, you can create the format "on-the-fly" using a SAS dataset containing the NDC_KEY and the LN. We won't discuss creating formats/informats "on-the-fly" in this lecture, but is discussed in the Advanced SAS presentation.

There are two types of SAS Formats: *Character Formats* and *Numeric Formats*. *Character formats* take character data and convert it to other character data. *Numeric formats* take numeric data and convert it to character data.

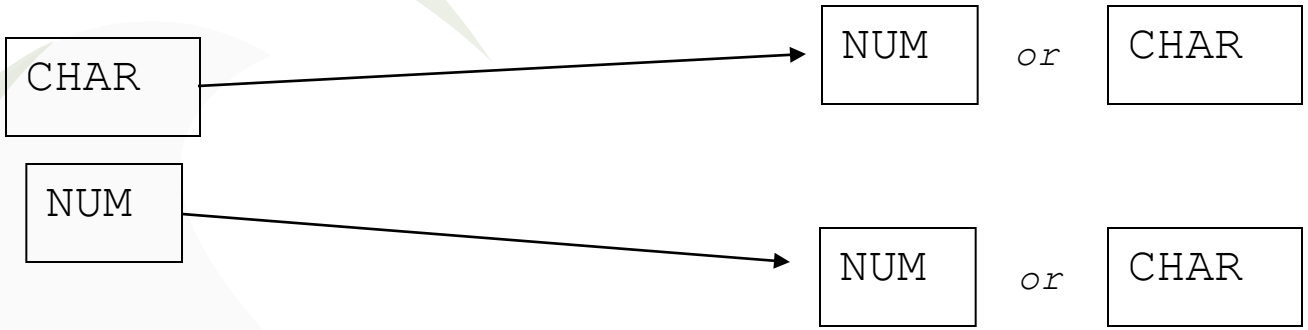
The FORMAT Procedure

sheepsqueezers.com

FORMATS



INFORMATS



Note: Use FORMAT to convert to characters instead of INFORMAT.

The FORMAT Procedure

SAS Formats

SAS Format – Convert Character to Character

Let's say we want to map NDC_KEY to its associated label name. Here's how to do that:

```
/* Character Format */
proc format;
 value $NDC "12345678901" = "PRILOSEC 10MG BOTTLE 100"
            "12345678902" = "PRILOSEC 20MG BOTTLE 100"
            "12345678903" = "PRILOSEC 30MG BOTTLE 100"
            "12345678904" = "PRILOSEC 40MG BOTTLE 100"
            other         = "UNKNOWN LABEL NAME      "
 ;
run;

data MyData;
 NDC_KEY="12345678901";LabelName=put(NDC_KEY,$NDC.);output;
 NDC_KEY="99999999999";LabelName=put(NDC_KEY,$NDC.);output;
run;

proc print data=MyData;
 var NDC_KEY LabelName;
run;
```

Obs	NDC_KEY	LabelName
1	12345678901	PRILOSEC 10MG BOTTLE 100
2	99999999999	UNKNOWN LABEL NAME

The FORMAT Procedure

SAS Formats

SAS Format – Convert Number to Character

Next, let's say we want to map AGE_KEY ranges to highly descriptive text. Here's how to do that:

```
/* Numeric Format */
proc format;
 value AGE  0-12   = "Prepubescent            "
           13-18   = "Just Trouble            "
           19-62   = "Working Stiff           "
           63-high = "Knocking on Death`s Door"
           other   = "Unknown Age             "
 ;
run;

data MyAgeData;
 AGE_KEY=6;AgeDesc=put(AGE_KEY,AGE.);output;
 AGE_KEY=15;AgeDesc=put(AGE_KEY,AGE.);output;
 AGE_KEY=52;AgeDesc=put(AGE_KEY,AGE.);output;
 AGE_KEY=67;AgeDesc=put(AGE_KEY,AGE.);output;
 AGE_KEY=-1;AgeDesc=put(AGE_KEY,AGE.);output;
 AGE_KEY=.;AgeDesc=put(AGE_KEY,AGE.);output;
run;

proc print data=MyAgeData;
 var AGE_KEY AgeDesc;
run;
```

Obs	AGE_KEY	AgeDesc
1	6	Prepubescent
2	15	Just Trouble
3	52	Working Stiff
4	67	Knocking on Death`s Door
5	-1	Unknown Age
6	.	Unknown Age

The FORMAT Procedure

SAS Informats

SAS Informats take character and numeric data and convert it to other character and numeric data. Unlike SAS Formats, SAS Informats use the `INVALUE` keyword instead of the `VALUE` keyword. Similar to SAS Character Formats, SAS Character Informats names must be preceded with a dollar-sign ($).

SAS Numeric Informat – Convert Number to Number

In the following example, we create a SAS Numeric Informat which converts one set of numbers (`AGE_KEY`s) to another set of numbers (`AgeGroup`).

```
proc format;
 invalue AGEGRP  0-<13  = 1 /* Prepubescent  */
                13-<19  = 2 /* Just Trouble */
                19-<63  = 3 /* Working Stiff */
                63-high = 4 /* Knocking on Death`s Door */
                other   = 5 /* Unknown Age */
 ;
run;

data MyAgeData;
 AGE_KEY=12;
 AgeGroup=input(AGE_KEY,AGEGRP.);
 output;
 AGE_KEY=25;
 AgeGroup=input(AGE_KEY,AGEGRP.);
run;
```

Obs	AGE_KEY	AgeGroup
1	12	1
2	25	3

The FORMAT Procedure

SAS Informats

SAS Numeric Informat – Convert Character to Number

Here is how you use the SAS Numeric Informat `PGRP`:

```
proc format;
 invalue $PGRP "12345678901" = 1 /* PRODUCT GROUP #1 */
               "12345678902" = 1 /* PRODUCT GROUP #1 */
               "12345678903" = 2 /* PRODUCT GROUP #2 */
               "12345678904" = 2 /* PRODUCT GROUP #2 */
               other         = 3 /* PRODUCT GROUP #3 */
 ;
run;

data MyNDCData;
 NDC_KEY="12345678901";ProdGroup=input(NDC_KEY,$PGRP.);output;
 NDC_KEY="12345678904";ProdGroup=input(NDC_KEY,$PGRP.);output;
 NDC_KEY="99999999999";ProdGroup=input(NDC_KEY,$PGRP.);output;
run;
```

Obs	NDC_KEY	ProdGroup
1	12345678901	1
2	12345678904	2
3	99999999999	3

The TRANSPOSE Procedure

If it's not obvious enough from the name of the procedure, the SAS Transpose procedure transposes data in your dataset. Thus, you can transpose rows to columns and vice versa.

All jocularity aside, if you don't set up the PROC TRANSPOSE correctly, you could wind up with data values placed the wrong columns! So, be careful and check the transposed (outgoing) dataset against the original (incoming) dataset!

Let's say that your data looks like this:

Obs	MONTH_OF_ SERVICE	PATIENT_ KEY	AVG_ COPAY_ AMT
1	JAN07	1111	10
2	MAR07	1111	30
3	JAN07	2222	40
4	FEB07	2222	50
5	MAR07	2222	60
6	JAN07	3333	70
7	FEB07	3333	80
8	MAR07	3333	90

Note how the MONTH_OF_SERVICE is going downward. Using PROC TRANSPOSE, we can create columns out of JAN07, FEB07 and MAR07! Note that, in this case, MONTH_OF_SERVICE is a character variable and not a SAS Date Variable with a format on it!!

The TRANSPOSE Procedure


```
proc transpose data=PatientKeyAvgCopayAmt out=T_PatientKeyAvgCopayAmt;
 by PATIENT_KEY;
 id MONTH_OF_SERVICE;
 var AVG_COPAY_AMT;
run;
```

Obs	PATIENT_KEY	_NAME_	JAN07	FEB07	MAR07
1	1111	AVG_COPAY_AMT	10	.	30
2	2222	AVG_COPAY_AMT	40	50	60
3	3333	AVG_COPAY_AMT	70	80	90

Take note that the MONTH_OF_SERVICE column was transposed as columns. Also note that the ID statement in the PROC TRANSPOSE specifies exactly which variable is to be transposed. That is, if MONTH_OF_SERVICE is a SAS Date Variable with the format *monyy5.* (for example, JAN05,FEB06, etc.) on it, then PROC TRANSPOSE will create the same variables as you see above. If MONTH_OF_SERVICE has the format *mmddyy10.*, then PROC TRANSPOSE will not create JAN07, but 01/01/2007. This is harder to work with! If you have no format on MONTH_OF_SERVICE, then PROC TRANSPOSE will create the columns using the actual date values such as 17167. Again, this is harder to work with! It's best to "create" this variable in the form that you want rather than let SAS create it.

Take note of the missing value for FEB07 for PATIENT_KEY=1111. Let's take a look at what happens if you do not use the ID statement on the PROC TRANSPOSE.

The TRANSPOSE Procedure


```
proc transpose data=PatientKeyAvgCopayAmt out=T_PatientKeyAvgCopayAmt;
 by PATIENT_KEY;
 var AVG_COPAY_AMT;
run;
```

Obs	PATIENT_KEY	_NAME_	COL1	COL2	COL3
1	1111	AVG_COPAY_AMT	10	30	.
2	2222	AVG_COPAY_AMT	40	50	60
3	3333	AVG_COPAY_AMT	70	80	90

Take note that the columns are now named COL1, COL2 and COL3. Also, take note that the "30" which is really for MAR2007 is now in the same column as the "50" and "80" which are for FEB2007. Without the ID statement, SAS will fill in as much as possible from left to right without regard to the original columns. But, don't fear this, because there are some places where this could come in handy. For example, assume we pulled RX data with hundreds of NDC codes. We can either use PROC TRANSPOSE specifying ID NDC_VAR (say, "V"||NDC_KEY) and we will get hundreds of columns, one for each NDC_KEY…most of which will be blank because not every patient uses all of those drugs. Assuming that the maximum number of drugs used by a single patient is 10, if you don't specify the ID statement, you will get 10 columns (not hundreds) in your transposed dataset. And we all know that small datasets are happy datasets!

The CONTENTS Procedure

So you have a SAS dataset, or maybe a LIBNAME set up to the location of all of your favorite SAS datasets…so how can you get a list of all of the datasets and what variables, number of rows, index information, etc. in each one of them? Why, use the PROC CONTENTS procedure, of course. Here is an example for a specific dataset using DATA=.

```
proc contents data=PatientKeyAvgCopayAmt;
run;

The CONTENTS Procedure

Data Set Name        WORK.PATIENTKEYAVGCOPAYAMT            Observations          8
Member Type          DATA                                  Variables             4
Engine               V9                                    Indexes               0
Created              Tuesday, July 24, 2007 08:57:27 AM    Observation Length    32
Last Modified        Tuesday, July 24, 2007 08:57:27 AM    Deleted Observations  0
Protection                                                 Compressed            NO
Data Set Type                                              Sorted                NO
Label
Data Representation  WINDOWS_64
Encoding             wlatin1  Western (Windows)

…continued…
```

The CONTENTS Procedure

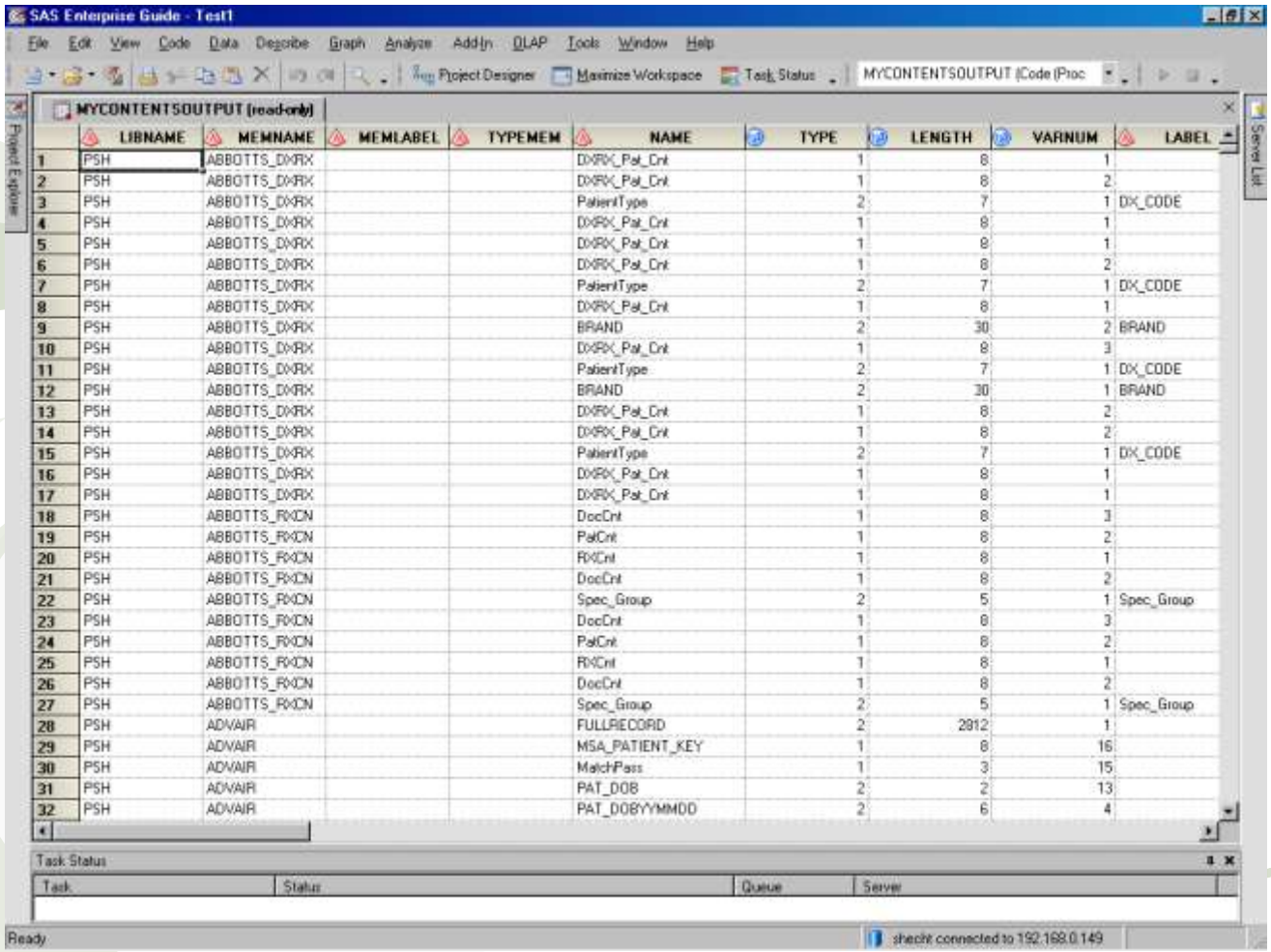

```
Engine/Host Dependent Information

Data Set Page Size          4096
Number of Data Set Pages    1
First Data Page             1
Max Obs per Page            126
Obs in First Data Page      8
Number of Data Set Repairs  0
File Name                   W:\SASWORK\_TD12328\Prc2\patientkeyavgcopayamt.sas7bdat
Release Created             9.0101M3
Host Created                W64_ASRV


  Alphabetic List of Variables and Attributes

#    Variable            Type    Len    Format

4    AVG_COPAY_AMT       Num       8
1    MONTH_OF_SERVICE    Char      6
3    PATIENT_KEY         Num       8
2    sdv                 Num       8    MONYY5.
```

Take note that the "#" column represents the order the variables were created in the dataset, so MONTH_OF_SERVICE was created first, whereas AVG_COPAY_AMT was created last. The variables are ordered in alphabetical order rather than by the # column. Use the PROC CONTENTS VARNUM option to order by #.

The CONTENTS Procedure

There may be times when you want to output the results of the CONTENTS procedure to a SAS dataset. To do this specify both the NOPRINT and the OUT= options on the PROC CONTENTS line:

```
proc contents data=PSH._all_ noprint out=MyContentsOutput;
run;
```



	LIBNAME	MEMNAME	MEMLABEL	TYPEMEM	NAME	TYPE	LENGTH	VARNUM	LABEL
1	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	1	
2	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	2	
3	PSH	ABBOTTS_DXRX			PatientType	2	7	1	DX_CODE
4	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	1	
5	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	1	
6	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	2	
7	PSH	ABBOTTS_DXRX			PatientType	2	7	1	DX_CODE
8	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	1	
9	PSH	ABBOTTS_DXRX			BRAND	2	30	2	BRAND
10	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	3	
11	PSH	ABBOTTS_DXRX			PatientType	2	7	1	DX_CODE
12	PSH	ABBOTTS_DXRX			BRAND	2	30	1	BRAND
13	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	2	
14	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	2	
15	PSH	ABBOTTS_DXRX			PatientType	2	7	1	DX_CODE
16	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	1	
17	PSH	ABBOTTS_DXRX			DXRX_Pat_Cnt	1	8	1	
18	PSH	ABBOTTS_RXCN			DocCnt	1	8	3	
19	PSH	ABBOTTS_RXCN			PatCnt	1	8	2	
20	PSH	ABBOTTS_RXCN			RXCnt	1	8	1	
21	PSH	ABBOTTS_RXCN			DocCnt	1	8	2	
22	PSH	ABBOTTS_RXCN			Spec_Group	2	5	1	Spec_Group
23	PSH	ABBOTTS_RXCN			DocCnt	1	8	3	
24	PSH	ABBOTTS_RXCN			PatCnt	1	8	2	
25	PSH	ABBOTTS_RXCN			RXCnt	1	8	1	
26	PSH	ABBOTTS_RXCN			DocCnt	1	8	2	
27	PSH	ABBOTTS_RXCN			Spec_Group	2	5	1	Spec_Group
28	PSH	ADVAIR			FULLRECORD	2	2812	1	
29	PSH	ADVAIR			MSA_PATIENT_KEY	1	8	16	
30	PSH	ADVAIR			MatchPass	1	3	15	
31	PSH	ADVAIR			PAT_DOB	2	2	13	
32	PSH	ADVAIR			PAT_DOBYYMMDD	2	6	4	

The DATASETS Procedure

The DATASETS procedure manages SAS datasets and allows you to rename SAS datasets, change column names, repair SAS datasets, delete SAS datasets, append multiple SAS datasets together, list PROC CONTENTS like information, create/delete indexes, and much, much more!

The DATASETS procedure has the concept of a RUN-Group. That is, you can perform multiple tasks with PROC DATASETS just by separating individual tasks with the RUN; keyword. You can end your PROC DATASETS procedure by using the QUIT; keyword at the end of your RUN-Groups.

For example, below I delete the temporary SAS dataset MyTempDataset, I then rename the dataset MyOldDataset to MyNewDataset, then I rename the variable MONTH_OF_SERVICE in MyNewDataset to SVC_MNTH:

```
proc datasets library=work;
 delete MyTempDataset;
run;
 change MyOldDataset=MyNewDataset;
run;
 modify MyNewDataset;
  rename MONTH_OF_SERVICE=SVC_MNTH;
run;
quit;
```

On the next page, we show part of the SAS Log:

The DATASETS Procedure


```
43          proc datasets library=work nolist;
44           delete MyTempDataset;
45          run;

NOTE: Deleting WORK.MYTEMPDATASET (memtype=DATA).
46           change MyOldDataset=MyNewDataset;
47          run;

NOTE: Changing the name WORK.MYOLDDATASET to WORK.MYNEWDATASET (memtype=DATA).
48           modify MyNewDataset;
49            rename MONTH_OF_SERVICE=SVC_MNTH;
NOTE: Renaming variable MONTH_OF_SERVICE to SVC_MNTH.
50          run;

NOTE: MODIFY was successful for WORK.MYNEWDATASET.DATA.
51          quit;

NOTE: PROCEDURE DATASETS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

The DATASETS Procedure

APPEND Statement

You can use PROC DATASETS to append one dataset, called the *data* dataset, to the end of another dataset, called the *base* dataset. This is much faster than a DATA SET with the SET command! Below we append the dataset MyExtraDataset into the dataset MyEntireDataset:

```
proc datasets library=work nolist;
 append base=MyEntireDataset data=MyExtraDataset;
run;
quit;
```

The great thing about using PROC DATASETS APPEND is that if the *base* dataset does not exist, it will be created from the *data* dataset when the APPEND Statement is run! Please don't use a DATA Step to do this!!

The DATASETS Procedure

CHANGE Statement

You can use PROC DATASETS to change the name of a dataset to another name. Below we change the name of the dataset MyExtraDataset into the dataset MyXtraDataset:

```
proc datasets library=work nolist;
 change base=MyExtraDataset data=MyXtraDataset;
run;
quit;
```

Again, don't use a DATA Step to do this if all you want to do is rename the dataset!

The DATASETS Procedure

DELETE Statement

You can use PROC DATASETS to delete datasets. Below we delete the dataset MyExtraDataset and MyXtraDataset:

```
proc datasets library=work nolist;
 delete MyExtraDataset MyXtraDataset;
run;
quit;
```

It's good practice to delete your WORK datasets after you are done with them – during the course of your SAS code – instead of waiting for them to be deleted automatically at the end of your SAS session.

The DATASETS Procedure

MODIFY Statement

The PROC DATASETS MODIFY statement allows you to perform several actions on a SAS dataset such as change formats/informats for one or more variables in the SAS dataset, change variable labels, rename variables, create/drop indexes, etc.

```
proc datasets library=work nolist;
 modify MyXtraDataset(label="This is my best dataset!");
  format PATIENT_PAY_AMT $dollar12.;
  label PATIENT_PAY_AMT="Sucker Payment Amount"
        COPAY_AMT="Doctor Lunch Money";
  index create PATIENT_KEY;
  index create IX_PATKEY_NDCKEY=(PATIENT_KEY NDC_KEY);
run;
 contents;
run;
quit;
```

Above, we modify the dataset MyXtraDataset by changing the format on the variable PATIENT_PAY_AMT, changing the labels for the PATIENT_PAY_AMT and COPAY_AMT variables, and creating one simple index and one composite index.

Finally, we use the CONTENTS statement to get a PROC CONTENTS-like listing of the dataset.

To delete an index, use the MODIFY INDEX DELETE statement.

The DATASETS Procedure

REPAIR Statement

There are some instances where your SAS dataset can get messed up. One example is if you delete a SAS index file (.sas7bndx) and then try to use the dataset. You will then have to issue the REPAIR statement so that SAS will re-create the index. If you want to remove an index, use the MODIFY INDEX DELETE statement.

```
proc datasets library=work nolist;
 repair MyXtraDataset;
run;
quit;
```

The DATASETS Procedure

COPY Statement

You can copy SAS datasets from one SAS library to another SAS library using the PROC DATASETS COPY Statement rather than using a DATA Step.

```
proc datasets library=work nolist;
 copy IN=WORK OUT=PSH;
  select MyXtraDataset;
run;
quit;
```

SAS System Options

SAS System Options

SAS provides a whole slew of system options that you can use to change the behavior of your SAS session. In this section, we outline several of the more useful options of which you should be aware. On the other hand, those options which aren't very useful, we won't spend any time on at all and won't even mention their existence.

Just to be clear, SAS System Options are switched on or off by placing the option on an `options` line at the top of your program, like so:

```
options ls=132 ps=60 mlogic symbolgen mprint;
run;
```

Note that some options are switched off by placing the word `NO` in front of the option or yelling at your monitor very loudly.

One important note is that there are some options which you should avoid changing completely. For example, the `CPUCOUNT` is already specified in the SAS Configuration File (`sasv9.cfg`) and should not be changed since this will affect the performance of those procedures which are thread-enabled (like `PROC SORT`, etc.).

- `CENTER | NOCENTER` – Specifies whether procedure output is centered on the page or not. Default: `NOCENTER`
- `COMPRESS=NO | YES | BINARY | CHAR` – Specifies SAS compresses a SAS dataset. `NO` indicates that the data in a SAS dataset are uncompressed. `YES` or `CHAR` attempts to compress repeated consecutive character data. `BINARY` attempts to compress each row in the SAS dataset that contains a lot of numeric variables. Default: `COMPRESS=NO`

SAS System Options

- `DATE | NODATE` – Specifies whether or not the date at which the SAS job began is printed in the log and listing files. DEFAULT: `NODATE`
- `ERRORS=#` – Specifies the maximum number of observations to print to the log if errors occur. After that, processing continues but no more messages will be printed to the log file. Default: `ERRORS=20`.
- `LINESIZE=#` – Specifies the linesize used by SAS Procedure output to the SAS listing file. The maximum is 256.
- `MISSING="char"` – Specifies what character to use to represent a missing value in SAS output. Default: .
- `MLOGIC | NOMLOGIC` – Specifies whether or not SAS macro logic is printed to the log file. Default: `NOMLOGIC`
- `MPRINT | NOMPRINT` – Specifies whether or not to print SAS statements generated by the macro processor. Default: `NOMPRINT`
- `NUMBER | NONUMBER` – Specifies whether or not to print page numbers to the SAS Listing file. Default: `NUMBER`
- `OBS=#` – Specifies the number of rows to read in from a SAS dataset or external file. Default: `OBS=MAX`. To check the syntax of your SAS program, use `OPTIONS OBS=0 NOREPLACE;`.
- `PAGENO=#` – Resets the page numbering in SAS output.
- `PAGESIZE = #` – Specifies the number of lines there are on a page before SAS ejects the page.

SAS System Options

- `SKIP = #` – Specifies the number of lines to skip at the top of SAS output.
- `SYMBOLGEN | NOSYMBOLGEN` – Specified whether or not SAS writes a message about the resolution of SAS macro variables to the SAS Log. Default: `NOSYMBOLGEN`.
- `YEARCUTOFF = #` – See the section on SAS Dates and Times in this deck.
- `MLOGICNEST | NOMLOGICNEST` – Specifies whether or not SAS will display nesting of SAS macros in the `MLOGIC` output.
- `MPRINTNEST | NOMPRINTNEST` – Specifies whether or not SAS will display nesting of SAS macros in the `MPRINT` output.

SAS Dataset Options

SAS Dataset Options

SAS provides a whole slew of options specifically for SAS datasets. These options, unlike SAS System Options, are placed in parentheses after the name of the dataset. In this section, we outline several of the more useful dataset options of which you should be aware.

Just to be clear, SAS Dataset Options are switched on or off by placing the option in parentheses after the name of the dataset, like so:

```
data bob;
 set wilma(obs=50);
run;
```

- `COMPRESS=NO | YES | BINARY | CHAR` – Specifies SAS compresses a SAS dataset. `NO` indicates that the data in a SAS dataset are uncompressed. `YES` or `CHAR` attempts to compress repeated consecutive character data. `BINARY` attempts to compress each row in the SAS dataset that contains a lot of numeric variables. Default: `COMPRESS=NO`
- `DROP=`*`variable-names`* – Specifies which variables to remove from the dataset.
- `FIRSTOBS=#` – Specifies which observation SAS processes first. Default: 1
- `IN=`*`variable`* – Creates a variable named *`variable`* that indicates whether the dataset contributed data to the current observation.

SAS Dataset Options

- `KEEP=variable-names` – Specifies which variables to keep for processing in an input dataset, or which variables to keep in the outgoing dataset.
- `LABEL='text-string'` – Specifies descriptive information for a dataset. Maximum length of `text-string` is 256 characters.
- `OBS=#` – Specifies which observation SAS processes last. Default: `OBS=MAX`
- `RENAME=(old-var-1=new-var-1 …)` – Renames a variable in a SAS dataset. Note that `DROP=` and `KEEP=` are applied before `RENAME=` if they occur together.
- `SORTEDBY=sorted-var-1 sorted-var-2…` – Use this option to let SAS know that an external file/database table being read in to a SAS dataset is already sorted. For example, if you are pulling data back from Oracle and use an ORDER BY clause in the Oracle SQL query, use the `SORTEDBY=` option on the SAS dataset you are creating to let SAS know that the dataset is already sorted. This will prevent `PROC SORT` from automatically sorting your dataset if it is already sorted.
- `WHERE=(where-clause)` – Specifies a WHERE clause on either input or output SAS datasets.

Support sheepsqueezers.com

If you found this information helpful, please consider supporting sheepsqueezers.com. There are several ways to support our site:

- ☐ Buy me a cup of coffee by clicking on the following link and donate to my PayPal account: Buy Me A Cup Of Coffee?.
- ☐ Visit my Amazon.com Wish list at the following link and purchase an item: http://amzn.com/w/3OBK1K4EIWIR6

Please let me know if this document was useful by e-mailing me at `comments@sheepsqueezers.com`.