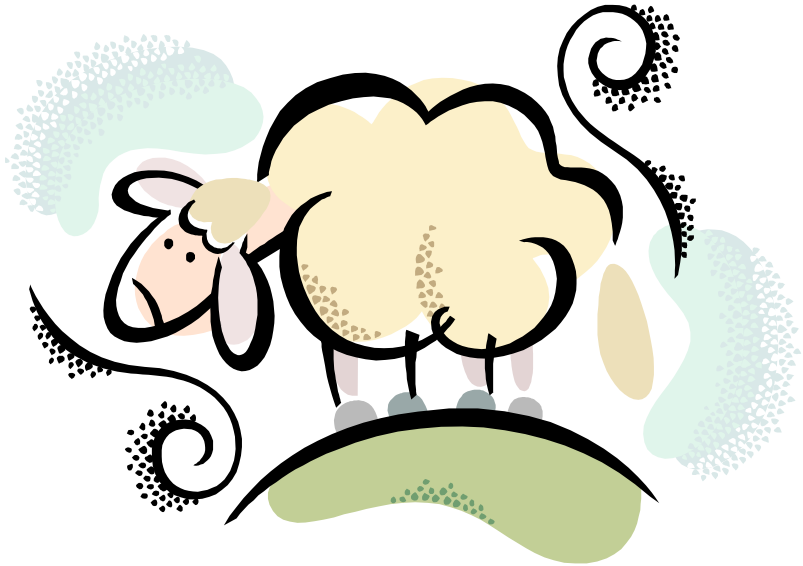


Advanced SAS

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

- Introducing Some New SAS Functions
- Introducing the `BAND()`,`BOR()`,`BXOR()`,`BNOT()`, `BLSHIFT()` and `BRSHIFT()` Functions
- Working with SAS Dates and Times
- Working with SAS Catalogs
- Using the Dictionary Tables in `PROC SQL`
- Creating a SAS Transport File
- Working with SAS Formats
- SAS Hash and Hash Iterator Objects
- SAS Output Delivery System
- Reading and Writing XML Data
- `FTP`, `URL` and `EMAIL` Options on the `FILENAME` Statement
- Working with Regular Expressions
- Using Indexes in SAS
- Locking and Unlocking SAS Datasets
- Generation Data Sets – It's All History, Man!
- Reading from and Writing to Microsoft Excel and Access
- Creating a New SAS Function Using the "C" Language
- Useful SAS System Options
- Useful SAS Dataset Options
- PROC-ology – New options for select procedures

Introducing Some New SAS Functions

Introducing Some New SAS Functions

LENGTH(arg), LENGTHC(arg), LENGTHM(arg), LENGTHN(arg)

The LENGTH() function will return the length of a character string after removing trailing blanks. One foible of this function is that if the character string is blank, the length returned will be 1, not 0.

```
data test1;
 myStr="Now is the time…   "; /* Total characters=19 */
 myLen=length(myStr); /* returns 16 because 3 trailing blanks removed */
run;
```

The LENGTHC() function will return the length of a character string INCLUDING the trailing blanks.

```
data test1;
 myStr="Now is the time…   "; /* Total characters=19 */
 myLen=lengthc(myStr); /* returns 19 */
run;
```

If myStr contains only blanks, then the LENGTHC() function will return the length of the character string with the blanks.

```
data test1;
 myStr="         "; /* Total blanks=9 */
 myLen=lengthc(myStr); /* returns 9  */
run;
```

If myStr contains no blanks, then LENGTHC() returns 1, not zero.

Introducing Some New SAS Functions

LENGTH(arg), LENGTHC(arg), LENGTHM(arg), LENGTHN(arg)

The LENGTHM() function will return the amount of memory allocated to a character string. Normally, this is equivalent to the number of characters in the character string. If you used a LENGTH or ATTRIB statement to set the default string length, LENGTHM() will return this value instead.

```
data test1;
 myStr="Now is the time…   "; /* Total characters=19 */
 myLen=lengthm(myStr); /* returns 19 */
run;

data test2;
 length myStr $ 100;
 myStr="Now is the time…   "; /* Total characters=19 */
 myLen=lengthm(myStr); /* returns 100 */
run;
```

The LENGTHN() function will return the length of a character string, excluding the trailing blanks. This function will return zero if the string is blank.

```
data test1;
 myStr="Now is the time…   "; /* Total characters=19 */
 myLen=lengthn(myStr); /* returns 16 */
 myStr2=""; /* Total blanks=0 */
 myLen2=lengthn(myStr2); /* Returns 0 */
 myLen3=length(myStr2); /* Returns 1 */
run;
```

Introducing Some New SAS Functions

COUNT(str,substr,mod), COUNTC(str,chars,mod)

The COUNT() function counts the number of times a substring *substr* appears in the string *str*.

```
data test1;
 myStr="The grinch bought an inch-long ruler in a pinch!";
 myCount=count(myStr,"inch"); /* returns 3 */
run;
```

You can ignore the case by using the "i" mod:

```
data test1;
 myStr="The grInch bought an Inch-long ruler in a pInch!";
 myCount=count(myStr,"inch","i"); /* returns 3 - "i" tells SAS to ignore case */
run;
```

The COUNTC() function counts the individual number of characters *chars* that appear in the string *str*.

```
data test1;
 myStr="The grinch bought an inch-long ruler in a pinch!";
 myCount=countc(myStr,"in"); /* returns 10 */
run;
```

You can count the characters that don't appear in chars by using the "v" mod:

```
data test1;
 myStr="The grinch bought an inch-long ruler in a pinch!";
 myCount=countc(myStr,"in","v"); /* returns 38 */
run;
```

Introducing Some New SAS Functions

SCAN(str,n,delimiters), SCANQ(str,n,delimiters)

The SCAN() function returns the nth word in *str* based on the delimiters.

```
data test1;
 myStr="The grinch bought an inch-long ruler in a pinch!";
 GrinchWord=scan(myStr,2," "); /* returns: grinch */
run;
```

```
data test1;
 myStr="The grinch bought an inch-long ruler in a pinch!";
 Words=scan(myStr,2,"-"); /* returns: long ruler in a pinch! */
run;
```

Note that you can specify more than one delimiter. The default delimiters are:

blank . < (+ & ! $ *); ^ – / , % |

The SCANQ() function is very similar to the SCAN() function except that it will ignore any delimiters appearing within quotes:

```
data test1;
 myStr="The grinch bought an 'inch-long' ruler in a pinch!";
 Words1=scan(myStr,2,"-"); /* returns: long' ruler in a pinch!*/
 Words2=scanq(myStr,2,"-"); /* returns nothing since delimiter is in ticks */
run;
```

Note: SCAN() and SCANQ() will treat consecutive delimiters as one.

Introducing Some New SAS Functions

CAT(str1,…), CATT(str1,…), CATS(str1,…), CATX(del,str1,…)

The CAT() function concatenates all of the string variables together and is equivalent to using the || vertical pipes.

```
data test1;
 A="Now ";B=" is ";C="the ";D="time…";
 myStr=cat(A,B,C,D); /* equivalent to A||B||C||D: Now  is the time… */
run;
```

The CATT() function concatenates all of the string variables together after it has TRIM'ed each variable:

```
data test1;
 A="Now ";B=" is ";C="the ";D="time…";
 myStr=catt(A,B,C,D); /* equivalent to TRIM(A)||TRIM(B)||TRIM(C)||TRIM(D): Now isthetime… */
run;
```

The CATS() function concatenates all of the string variables together after it has LEFT'ed *and* TRIM'ed each variable:

```
data test1;
 A="Now ";B=" is ";C="the ";D="time…";
 myStr=cats(A,B,C,D); /* equivalent to LEFT(TRIM(A))||LEFT(TRIM(B))||LEFT(TRIM(C))||LEFT(
TRIM(D)): Nowisthetime… */
run;
```

Introducing Some New SAS Functions

CAT(str1,…), CATT(str1,…), CATS(str1,…), CATX(del,str1,…)

The `CATX()` function is the same as the `CATS()` function with the ability to specify a delimiter between the variables:

```
data test1;
 A="Now ";B=" is ";C="the ";D="time…";
 myStr=catx("-",A,B,C,D); /* equivalent to LEFT(TRIM(A))||"-"||LEFT(TRIM(B))||"-
"||LEFT(TRIM(C))||"-"||LEFT( TRIM(D)): Now-is-the-time… */
run;
```

Note: You can use the `OF var1-varN` syntax with the `CAT*` functions:

```
data test1;
 X1="Now ";X2=" is ";X3="the ";X4="time…";
 myStr=catx("-",OF X1-X4); /* Now-is-the-time… */
run;
```

Note: If one of your variables is blank, it is ignored:

```
data test1;
 X1="Now ";X2=" is ";X3="      ";X4="time…";
 myStr=catx("-",OF X1-X4); /* Now-is-time… */
run;
```

Introducing Some New SAS Functions

TRIM(), TRIMN()

The TRIM() function will remove trailing blanks from your character string. If your character string is nothing but blanks, it will return one blank. If your character string is missing, it will return one blank.

```
data test1;
 length C $ 1;
 A="Now is the time…     ";
 B="     ";
 TrimA=trim(A); /* Now is the time… */
 D="X" || trim(B) || "Y"; /* X Y */
 E="X" || trim(C) || "Y"; /* X Y */
run;
```

The TRIMN() function is the same as TRIM() except that a blank string will result in a null string (no blanks):

```
data test1;
 length C $ 1;
 A="Now is the time…     ";
 B="     ";
 TrimnA=trimn(A); /* Now is the time… */
 D="X" || trimn(B) || "Y"; /* XY */
 E="X" || trimn(C) || "Y"; /* XY */
run;
```

Introducing Some New SAS Functions

STRIP()

The `STRIP()` function removes leading and trailing blanks from your character string in a similar was as `TRIMN(LEFT))` does. If your character string is nothing but blanks, it will return zero blanks. If your character string is missing, it will return zero blanks (NULL). The `STRIP()` Function runs faster than `TRIMN` and `LEFT` put together.

```
data test1;
 length C $ 1;
 A="Now is the time…     ";
 B="     ";
 StripA=strip(A); /* Now is the time… */
 D="X" || strip(A) || "Y"; /* XNowisthetimeY */
 E="X" || strip(B) || "Y"; /* XY */
 F="X" || strip(C) || "Y"; /* XY */
run;
```

Introducing Some New SAS Functions

GETOPTION(option-name,mod,mod,…)

The GETOPTION() function will return the setting of a SAS System Option or SAS Graphics GOPTION.

```
options ps=132;
run;

data test1;
 pagesize=getoption("PS"); /* returns 132 as a character string */
 pagesizeOption=getoption("PS","KEYWORD"); /* returns PS=132 */
run;
```

As an example of using the SAS/Graph GOPTIONS:

```
goptions hsize=11 inches;
run;

data test1;
 hsize=getoption("hsize"); /* returns: 11.0000 */
 hsizeGoption=getoption("hsize","IN","KEYWORD"); /* returns: HSIZE=11.0000 in. */
run;
```

Introducing Some New SAS Functions

FIND(str,substr,startpos,mods), FINDC(str,chars,startpos,mods)

The FIND() function searches the string *str* for the string *substr* starting at *startpos* and returns the starting position of *substr* if found within *str*. You can ignore case and/or trailing blanks using *mods*.

```
data test1;
 str="Now is the time for all good men to come to the aide of their country!";
 substr="for all good";
 MyPOS=find(str,substr,1); /* returns 17 */
run;
```

You can ignore case by specifying the mods "i":

```
data test1;
 str="Now is the time foR All gOOd men to come to the aide of their country!";
 substr="for all good";
 MyPOS=find(str,substr,1,"i"); /* returns 17 */
run;
```

The FINDC() function searches the string *str* for the characters *chars* starting at *startpos* and returns the starting position of a character appearing in *chars*. If not found, zero is returned.

```
data test1;
 str="Now is the time for all good men to come to the aide of their country!";
 chars="abc";
 MyPOS=findc(str,chars,1); /* returns 21 (the "a") */
run;
```

Introducing Some New SAS Functions

FACT(n), PERM(n,r), COMB(n,r), CALL ALLPERM(k,var1,…,varN)

The FACT() function returns the factorial of *n*: n!

```
data test1;
 factN=fact(4); /* returns 24 */
run;
```

The PERM() function returns the number of **permutations** of *n* things taken *r* at a time: P(n,r)=n!/(n-r)!

```
data test1;
 perm62=perm(6,2); /* 30 */
run;
```

The COMB() function returns the number of **combinations** of *n* things taken *r* at a time: C(n,r)=n!/r!(n-r)!

```
data test1;
 comb62=comb(6,2); /* 15 */
run;
```

The CALL ALLPERM(k,var1,…,varN) call routine creates all combinations of the data contained in var1,…,varN. Think of this as a way to create a full-factorial design matrix. In order to use this call routine, you must call it FACT(N) times each time replacing k with the next value. For example,

Introducing Some New SAS Functions

FACT(n), PERM(n,r), COMB(n,r), CALL ALLPERM(k,var1,…,varN)

```
data FullFactDesign;
 array X[4] $ 1 ('A','B','C','D');
 NFact=fact(4);
 do k=1 to NFact;
  call allperm(k,of X[*]);
  output;
 end;
run;

proc sort data=FullFactDesign;
 by X1 X2 X3 X4;
run;
```

Obs	X1	X2	X3	X4	NFact	k
1	A	B	C	D	24	1
2	A	B	D	C	24	2
3	A	C	B	D	24	8
4	A	C	D	B	24	7
5	A	D	B	C	24	3
6	A	D	C	B	24	6
7	B	A	C	D	24	24
8	B	A	D	C	24	23
9	B	C	A	D	24	17
10	B	C	D	A	24	18
11	B	D	A	C	24	22
12	B	D	C	A	24	19
13	C	A	B	D	24	9
14	C	A	D	B	24	10
15	C	B	A	D	24	16
16	C	B	D	A	24	15
17	C	D	A	B	24	11
18	C	D	B	A	24	14
19	D	A	B	C	24	4
20	D	A	C	B	24	5
21	D	B	A	C	24	21
22	D	B	C	A	24	20
23	D	C	A	B	24	12
24	D	C	B	A	24	13

Introducing Some New SAS Functions

CONSTANT()

The CONSTANT() function returns one of several mathematical constants such as PI and E.

```
data test1;
 PI=constant("PI"); /* returns 3.14159… */
 E=constant("E"); /* returns 2.71828… */
run;
```

Introducing Some New SAS Functions

IFC(boolean,true,false,missing) and IFN(boolean,true,false,missing)

Both the IFC() and IFN() functions are compact versions of an IF-THEN-ELSE statement, and similar to Oracle's DECODE and Excel's IIF functions.

```
data test1;
 length AgeDesc $ 10;
 AGE_KEY=25;
 AgeDesc=ifc(AGE_KEY<45,"Youngster","Oldster","Unknown");
 output;
 AGE_KEY=95;
 AgeDesc=ifc(AGE_KEY<45,"Youngster","Oldster","Unknown");
 output;
run;

data test2;
 length AgeGroup 3;
 AGE_KEY=25;
 AgeGroup=ifn(AGE_KEY<45,1,2,3);
 output;
 AGE_KEY=95;
 AgeGroup=ifn(AGE_KEY<45,1,2,3);
 output;
run;
```

Introducing
BAND(), BOR(),
BXOR(), BNOT(),
BLSHIFT() and BRSHIFT()

Introducing BAND(), BOR(), etc.

The `BAND()`, `BOR()`, `BXOR()`, `BNOT()`, `BLSHIFT()` and `BRSHIFT()` functions are introduced separately since they require the user to know a little bit about the binary number system. As you all know, on a day-to-day basis we all use the decimal number system, also know as Base 10. We learned in grade school about the one's column, the ten's column, the hundred's column, etc. Recall, for example, that we can break up the number 1234 like this:

$$1x1000 + 2x100 + 3x10 + 4x1 = 1234_{10}$$

which is equivalent to

$$1x10^3 + 2x10^2 + 3x10^1 + 4x10^0 = 1234_{10}$$

It shouldn't be a surprise to you all that we can use other bases besides 10. For example, computers use the binary number system (Base 2). You've also probably been exposed to the octal number system (Base 8) and the hexadecimal number system (Base 16) in your computer classes. We will concentrate on Base 2. Let's represent the decimal number 123_{10} as a binary number:

$$\begin{aligned} 123_{10} &= 1x2^6 + 1x2^5 + 1x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 \\ &= 1111011_2 \end{aligned}$$

How did I compute this? I used the Window's Calculator! Cheater!!

Introducing BAND(), BOR(), etc.

$$123_{10} = 1x2^6 + 1x2^5 + 1x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$$
$$= 1111011_2$$

Note that each column in the number 1111011_2 is called a "bit" and not a column. You can add binary numbers together just like you can decimal numbers:

123_{10}	1111011_2
$+ \quad 1_{10}$	$+ 0000001_2$
--------	----------
124_{10}	1111100_2

Note that $9_{10}+1_{10}=10_{10}$, $1_2+1_2=10_2$. There are other things you can do with the bits within binary numbers. First, you can *Logically AND* two bits together. Here is how you *Logically AND* two bits together:

A	B	A&B
0	0	0
0	1	0
1	0	0
1	1	1

Introducing BAND(), BOR(), etc.

Second, you can *Logically OR* two bits together. Here is how you *Logically OR* two bits together:

A	B	A|B
0	0	0
0	1	1
1	0	1
1	1	1

Third, you can *Logically XOR* (called exclusive OR) two bits together. Here is how you *Logically XOR* two bits together:

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

Finally, you can *Logically Not* a bit; that is, a zero becomes a one, and a one becomes a zero.

Introducing BAND(), BOR(), etc.

If you think that Logical AND, Logical OR, Logical XOR, and Logical Not are related to `BAND()`, `BOR()`, `BXOR()`, and `BNOT()`, you are right, and yet simultaneously wrong. The SAS functions `BAND()`, `BOR()`, `BXOR()`, and `BNOT()` work not only on just one bit between the two numbers, but all the bits between the numbers. The tables I showed above will help you to compute these functions. For example, let's Logically AND the numbers 15_{10} and 3_{10}:

15_{10}	1111_2	
& 3_{10}	& 0011_2	
-------	--------	`BAND(15,3)=3`
3_{10}	0011_2	

Let's Logically OR the numbers 15_{10} and 3_{10}:

15_{10}	1111_2	
| 3_{10}	| 0011_2	
-------	--------	`BOR(15,3)=15`
15_{10}	1111_2	

Let's Logically XOR the numbers 15_{10} and 3_{10}:

15_{10}	1111_2	
^ 3_{10}	^ 0011_2	
-------	--------	`BXOR(15,3)=12`
12_{10}	1100_2	

Introducing BAND(), BOR(), etc.

So far, we've seen how to Logically AND, OR, and XOR binary numbers together. Before we go any further, though, I want to show you how to display a SAS numerical value (Base 10) as a binary number (Base 2).

To display a SAS numeric value as a binary number, or to create a SAS character representation of a binary number using the `PUT()` function, you use the *`BINARYw.`* format:

```
data test1;
 DecNum=123;
 BinRepr=put(DecNum,binary7.); /* contains 1111011 as a character string */
run;
```

Note that if you use a wider width, say 12 instead of 7, SAS will zero fill from the left:

```
data test1;
 DecNum=123;
 BinRepr=put(DecNum,binary12.); /* contains 000001111011 as char string */
run;
```

Note: `BAND()`, `BOR()`, `BXOR()`, and `BNOT()` only work on numbers with a maximum of 32-bits. This is true for both 64-bit versions and 32-bit versions of SAS.

Introducing BAND(), BOR(), etc.

At this point, you all may be wondering why I'm teaching this to you. It may seem that it has no relevance to you. That's where you're wrong! I've used these ideas successfully when dealing with concomitancy such as with pharmaceuticals, credit card use, etc.

For concomitancy, I've seen programmers create long strings of text containing delimited drug names, say:

LEXXEL+LIPITOR+NEXIUM+VICODIN

This takes up a lot of space! Instead, you can assign each drug a unique number n=1,2,3,… and use the fact that 2^{n-1} flips a bit in an integer to create your concomitancy mapping. For example, let's assign each of the drugs above a unique integer and compute 2^{n-1}:

		binary
1=LEXXEL	$2^{n-1} = 2^0 = 1$	0001
2=LIPITOR	$2^{n-1} = 2^1 = 2$	0010
3=NEXIUM	$2^{n-1} = 2^2 = 4$	0100
4=VICODIN	$2^{n-1} = 2^3 = 8$	1000
	15	1111

Take note that each bit in the number 1111_2 indicates *one and only one* drug.

Introducing BAND(), BOR(), etc.

Here is an example of how to compute concomitancy using drugs.

Step One: Create List of Unique NDCs:

```
proc sort data=MY_DRUG_DATA(keep=NDC_KEY) out=UNIQUE_NDCS nodupkey;
 by NDC_KEY;
run;

data UNIQUE_NDCS;
 set UNIQUE_NDCS;
 ROWNUM=_n_;
 DRUG_PWR=2**(ROWNUM-1);
 BIN_REPR=put(DRUG_PWR,binary32.);
 DRUG_NAME="DRUG " || put(ROWNUM,z4.);
run;
```

The next slide shows example data. Note that the drug names are made up and appear as "DRUG ####".

Introducing BAND(), BOR(), etc.

Obs	NDC_KEY	ROWNUM	DRUG_PWR	BIN_REPR	DRUG_NAME
1	00008083701	1	1	00000000000000000000000000000001	DRUG 0001
2	00008084181	2	2	00000000000000000000000000000010	DRUG 0002
3	00045152550	3	4	00000000000000000000000000000100	DRUG 0003
4	00069215030	4	8	00000000000000000000000000001000	DRUG 0004
5	00074316560	5	16	00000000000000000000000000010000	DRUG 0005
6	00093073310	6	32	00000000000000000000000000100000	DRUG 0006
7	00093081001	7	64	00000000000000000000000001000000	DRUG 0007
8	00093100301	8	128	00000000000000000000000010000000	DRUG 0008
9	00093519505	9	256	00000000000000000000000100000000	DRUG 0009
10	00093714656	10	512	00000000000000000000001000000000	DRUG 0010
11	00172208380	11	1024	00000000000000000000010000000000	DRUG 0011
12	00172640744	12	2048	00000000000000000000100000000000	DRUG 0012
13	00186108805	13	4096	00000000000000000001000000000000	DRUG 0013
14	00186504031	14	8192	00000000000000000010000000000000	DRUG 0014
15	00378023205	15	16384	00000000000000000100000000000000	DRUG 0015
16	00378145201	16	32768	00000000000000001000000000000000	DRUG 0016
17	00378335299	17	65536	00000000000000010000000000000000	DRUG 0017
18	00406036005	18	131072	00000000000000100000000000000000	DRUG 0018
19	00406053201	19	262144	00000000000001000000000000000000	DRUG 0019
20	00781518101	20	524288	00000000000010000000000000000000	DRUG 0020
21	45802075930	21	1048576	00000000000100000000000000000000	DRUG 0021
22	50924038110	22	2097152	00000000001000000000000000000000	DRUG 0022
23	53265040610	23	4194304	00000000010000000000000000000000	DRUG 0023
24	53489053601	24	8388608	00000000100000000000000000000000	DRUG 0024
25	58177032418	25	16777216	00000001000000000000000000000000	DRUG 0025
26	61570011201	26	33554432	00000010000000000000000000000000	DRUG 0026
27	62856024630	27	67108864	00000100000000000000000000000000	DRUG 0027
28	64455079942	28	134217728	00001000000000000000000000000000	DRUG 0028
29	65597010730	29	268435456	00010000000000000000000000000000	DRUG 0029

Introducing BAND(), BOR(), etc.

Step Two: Create Distinct List of Patients and Drugs (VERY IMPORTANT):

```
proc sort data=MY_DRUG_DATA(keep=NDC_KEY PATIENT_KEY) out=PAT_NDC_DATA nodupkey;
 by NDC_KEY PATIENT_KEY;
run;
```

Step Three: Merge `PAT_NDC_DATA` with `UNIQUE_NDCS`:

```
data DRUG_CONCOM;
 merge PAT_NDC_DATA(in=inL) UNIQUE_NDCS(in=inR);
 by NDC_KEY;
 if inL & inR;
run;

proc means data=DRUG_CONCOM nway noprint;
 class PATIENT_KEY;
 var DRUG_PWR;
 output out=DRUG_CONCOM(drop=_type_ _freq_) sum(DRUG_PWR)=DRUG_CONCOM;
run;
```

The next slide shows example data.

Introducing BAND(), BOR(), etc.

Obs	PATIENT_KEY	DRUG_CONCOM
1	35089063	00000010000000000001000000000100
2	35089246	00000000000000000000000100000000
3	35090076	00000000100000000000001000000000
4	35090386	00001000000000000100000000000010
5	48005690	00000000000000000010000000000000
6	48005870	00000001000000000000000000000000
7	48007727	00000000000000001000000001000000
8	48008670	00000000000000000000000000001000
9	48008996	00000000001000000000000000000000
10	48009210	00000000000000000000000010000000
11	280458490	00000000000000000000010000000000
12	298987732	00000000010000000000000000000000
13	298987863	00000000000000000000000000010000
14	298988140	00000000000000010000000000000000
15	298988305	00000000000001000000000000000000
16	298988383	00000000000001000000000000000000
17	302917888	00000000000100000000000000000000
18	302918005	00000000000000000000100000000000
19	302918793	00000000000000000000000000100000
20	302918890	00010000000000000000000000000100
21	302919328	00000100000000000000000000000000
22	302920042	00000000000000000000000000000001
23	302920411	00000000000010000000000000000000

Introducing BAND(), BOR(), etc.

Step Four: Map DRUG_CONCOM to list of Drug Names

There are several ways to map DRUG_CONCOM to the list of corresponding drug names. We will discuss how to use SAS Format during the lecture on SAS Formats. Right now, we'll just use an IF-THEN statement and a DO-Loop.

```
data FINAL(drop=i);
 length ALL_DRUGS $ 1000;
 attrib DRUG_CONCOM format=binary32.;
 set DRUG_CONCOM;
 ALL_DRUGS="";
 do i=1 to 29;
  if band(DRUG_CONCOM,2**(i-1))>0 then do;
   ALL_DRUGS=catx("-",ALL_DRUGS,"DRUG " || put(i,z3.));
  end;
 end;
run;
```

The next slide shows example data.

Introducing BAND(), BOR(), etc.

Obs	PATIENT_KEY	DRUG_CONCOM	ALL_DRUGS
1	35089063	00000010000000000001000000000100	DRUG 003-DRUG 013-DRUG 026
2	35089246	00000000000000000000000100000000	DRUG 009
3	35090076	00000000100000000000001000000000	DRUG 010-DRUG 024
4	35090386	00001000000000000100000000000010	DRUG 002-DRUG 015-DRUG 028
5	48005690	00000000000000000010000000000000	DRUG 014
6	48005870	00000001000000000000000000000000	DRUG 025
7	48007727	00000000000000001000000001000000	DRUG 007-DRUG 016
8	48008670	00000000000000000000000000001000	DRUG 004
9	48008996	00000000001000000000000000000000	DRUG 022
10	48009210	00000000000000000000000010000000	DRUG 008
11	280458490	00000000000000000000010000000000	DRUG 011
12	298987732	00000000010000000000000000000000	DRUG 023
13	298987863	00000000000000000000000000010000	DRUG 005
14	298988140	00000000000000010000000000000000	DRUG 017
15	298988305	00000000000001000000000000000000	DRUG 019
16	298988383	00000000000000100000000000000000	DRUG 018
17	302917888	00000000001000000000000000000000	DRUG 021
18	302918005	00000000000000000000100000000000	DRUG 012
19	302918793	00000000000000000000000000100000	DRUG 006
20	302918890	00010000000000000000000000000100	DRUG 003-DRUG 029
21	302919328	00000100000000000000000000000000	DRUG 027
22	302920042	00000000000000000000000000000001	DRUG 001
23	302920411	00000000000010000000000000000000	DRUG 020

Introducing BAND(), BOR(), etc.

As another example of how to use binary numbers to solve a common problem, let's determine the number patients who appear in the DATA1 dataset, the DATA2 dataset, the DATA3 dataset, the DATA1/DATA2 datasets intersected, the DATA1/DATA3 datasets intersected, and so on. One way to do this is to create a list of all patients from the DATA1, DATA2 and DATA3 datasets, then count the number of patients in each set separately, or after performing inner joins between the sets two-at-a-time then three-at-a-time. Why is this not a good idea?

Another way to do this is to use your bits. Here is some SQL code:

```
SELECT ALL_SETS,COUNT(*) as NUM_PATS
 FROM (
        SELECT PATIENT_KEY,SUM(SET_INDICATOR) as ALL_SETS
         FROM (
               SELECT DISTINCT PATIENT_KEY,1 AS SET_INDICATOR
                FROM DATA1
               UNION
               SELECT DISTINCT PATIENT_KEY,2 AS SET_INDICATOR
                FROM DATA2
               UNION
               SELECT DISTINCT PATIENT_KEY,4 AS SET_INDICATOR
                FROM DATA3
              )
         GROUP BY PATIENT_KEY
      )
 GROUP BY ALL_SETS
 ORDER BY 1
```

Introducing BAND(), BOR(), etc.

Below are the results of the SQL query on the previous slide. Take note that the column ALL_SETS indicates, when displayed in binary format (as shown after the equal sign below), what each row means. For example, when ALL_SETS=3, represented as 011 in binary, the value 17797 is the number of unique patients who are in the DATA1 and DATA2 datasets simultaneously. Whereas when ALL_SETS=7, the value 681 is the number of unique patients who appear in all three datasets DATA1, DATA2 and DATA3.

ALL_SETS	NUM_PATS		
1	18316	= 001	(DATA1)
2	2676	= 010	(DATA2)
3	17797	= 011	(DATA1/DATA2)
4	1036	= 100	(DATA3)
5	5970	= 101	(DATA3/DATA1)
6	768	= 110	(DATA3/DATA2)
7	681	= 111	(DATA3/DATA2/DATA1)

Introducing BAND(), BOR(), etc.

Two Additional Functions: BLSHIFT and BRSHIFT

You may never need these two functions, but just in case, I mention them.

The `BLSHIFT(num1,shift)` function takes a number *`num1`* and performs a bitwise shift left *`shift`* number of times. A "shift left" one time takes all of the bits in `num1` and moves them over to the left one column. Column 1, the furthest to the right, is set to zero. Column 32, the furthest to the left, is dropped.

```
data test1;
 ANum=3;                          /* 0000011= 3 */
 ANum_Shift_1=blshift(ANum,1); /* 0000110= 6 */
 ANum_Shift_2=blshift(ANum,2); /* 0001100=12 */
run;
```

The `BRSHIFT(num1,shift)` function takes a number *`num1`* and performs a bitwise shift right *`shift`* number of times. A "shift right" one time takes all of the bits in `num1` and moves them over to the right one column. Column 1, the furthest to the right, is dropped. Column 32, the furthest to the left, is zero filled.

```
data test1;
 ANum=3;                          /* 0000011= 3 */
 ANum_Shift_1=brshift(ANum,1); /* 0000001= 1 */
 ANum_Shift_2=brshift(ANum,2); /* 0000000= 0 */
run;
```

Working with SAS Dates and Times

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.

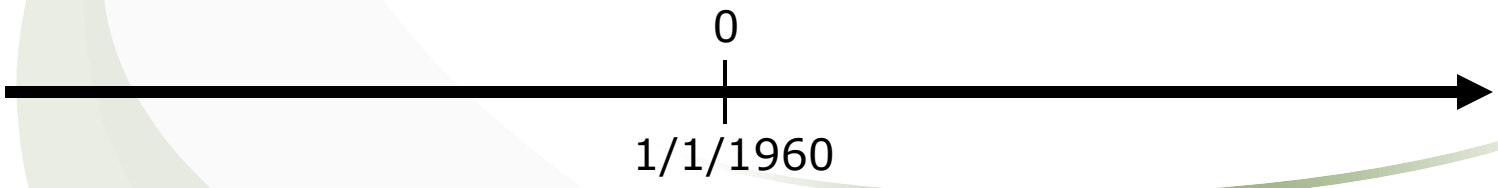
So, how does SAS represent dates and times?

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

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

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 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.

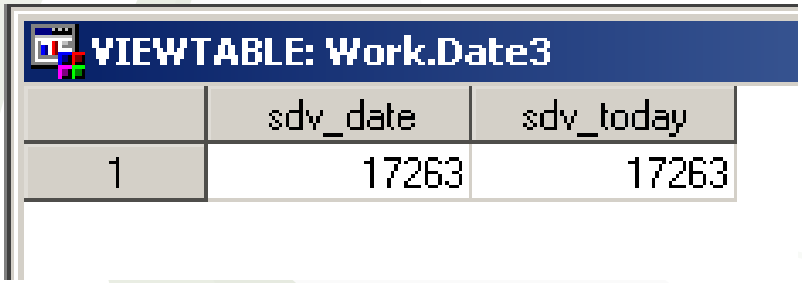
```
data date1;
 sdv=mdy(12,31,1990); /* SAS Date Value: 11322 */
run;
```

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 (MMDDYY10.):

```
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 while leaving the underlying values untouched.

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 – which always returns a character string – 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 or not.

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 (again, the underlying value is unchanged):

```
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.

```
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

Another nice function is the `JULDATE(sdv)` function used 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).

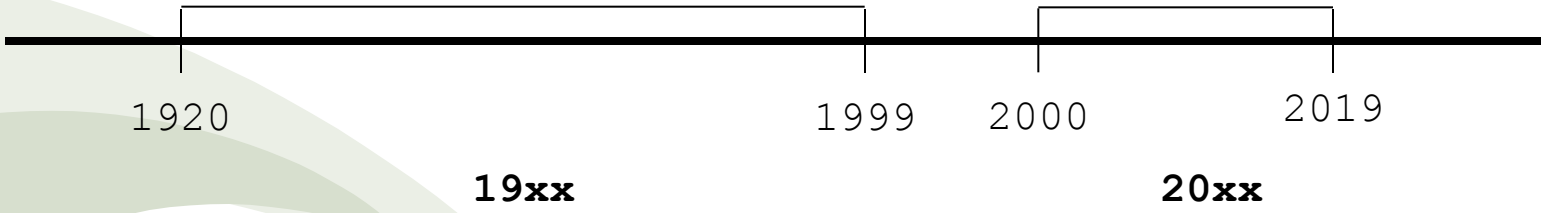
```
data date6;
 sdv_today=today(); /* 17263 */
 Hark_Who_Goes_There=juldate7(sdv_today); /* 2007097 */
run;
```

There is a rather important *SAS System Option* we should mention: `YEARCUTOFF`. Recall a few years ago there was this rather unjustifiable panic over the *Year 2000 problem*. You know, when the clocks turned from 12/31/1999:23:59:59 to 01/01/2000:00:00:00 and the entire world was supposed to end causing people to rush to the store and stock up on dozens of cans of yams to help them survive the impending nuclear winter. Obviously, we are still here…have you finished all of those yams yet? Me neither. The Y2K Problem was taken very seriously and many companies hired programmers to modify legacy code and check that their computer systems would work when the date rolled over. Why did this Y2K Problem happen? It's all due to space…or the lack of space. When programmers coded year values, instead of using four-digits, they used the last two digits to save space. So, 1999 became 99 and 1932 became 32. Sadly, 2000 became 00 which screwed up any date difference calculation such as 1/1/2000-1/1/1970=00-70=-70. Oops! Naturally, SAS had taken this into account and introduced the `YEARCUTOFF` system option. This option tells SAS what to do with a *two-digit year* in a SAS Date/Datetime Value.

Working with SAS Dates and Times

For example, if `YEARCUTOFF=1920`, the default, then any two-digit year within the range of 20 to 99 becomes 19xx, whereas any two-digit year within the range of 00 to 19 becomes 20xx. So, the year 34 is 1934, whereas the year 05 becomes 2005.

This has no effect if the year is four-digits. And, you guessed it, it's best to use four-digit years.



Since dates stored in Oracle and SQL Server databases contain full four-digit years, you don't have to worry too much about this option. Where you might have to think about it is if you're given, say, a text-file containing birthdates. If the year of birth is only 2-digits, you might want to shift the `YEARCUTOFF` option over a bit. For example, is a year of birth of *06* a very OLD patient (1906) or a very YOUNG patient (2006)?

Working with SAS Dates and Times

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;
```

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 are two SAS functions which allow you to shift SAS Date Values and SAS Datetime Values, as well as compute the differences between them.

Working with SAS Dates and Times

- 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.

```
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!

Working with SAS Dates and Times

SAS has shorthand notation to help you create SAS dates, datetimes and times quickly and easily.

- To create a SAS date, you can specify `'DDMONYYYY'd`
- To create a SAS datetime, you can specify `'DDMONYYYY:HH:MM:SS'dt`
- To create a SAS time, you can specify `'HH:MM:ss't`

Note that if you use these within a SAS macro, the SAS Macro processor will see them as text, so you will have to use an INPUTN function surrounded by a `%SYSFUNC()` function. Don't forget to remove the quotes! Here is an example:

```
%if %sysfunc(inputn(DDMONYYYY,date9.)) ge %sysfunc(inputn(DDMONYYYY,date9.))
```

Working with SAS Dates and Times

When pulling data from either an Oracle or SQL Server database table, be aware that SAS ALWAYS returns SAS Datetime Values for the DATE/DATETIME/ SMALLDATETIME data types! Recall that you can use the `DATEPART()` function to convert a SAS Datetime Value to a SAS Date Value within your SAS program if you plan on doing any date calculations.

If you have a SAS dataset with one or more SAS Date Values, and you want to load these SAS Date Values into either an Oracle or SQL Server database table, you **MUST** convert your SAS Date Values to SAS Datetime Values using the `DHMS()` function **AND** you must also specify a SAS Format of `DATETIME.` on each column in the SAS dataset you want loaded into the database. SAS/Access uses this SAS Format to determine what type of data is in the field (datetime, character, number). If you do not specify a format, SAS will most likely load the data in to the Oracle/SQL Server table incorrectly, at least for the datetime columns. Always check the data loaded into the database to ensure the dates are correct against the original data!

Working with SAS Dates and Times

Below is a detailed example. Note that we are showing SAS/Access to OLEDB code below, but other SAS/Access products will work similarly.

```
*-------------------------------------------------------------------------------*;
* Set up a LIBNAME to the database (Oracle) in order to INSERT data later.     *;
*-------------------------------------------------------------------------------*;
libname dbORCL oledb init_string="Provider=OraOLEDB.Oracle;Data Source=ORCL;USER
ID=SCOTT;PASSWORD=TIGER;PORT=1521";
run;

*-------------------------------------------------------------------------------*;
* Pull 1000 rows from DATA1 keeping DATE_AUTH1,PATIENT_KEY and NDC_KEY.        *;
* Use the DATEPART() function to convert DATE_AUTH1 to a SAS Date Value.       *;
*-------------------------------------------------------------------------------*;
proc sql noprint;
 connect to oledb as ORCL (init_string="Provider=OraOLEDB.Oracle;Data Source=ORCL;USER
ID=SCOTT;PASSWORD=TIGER;PORT=1521");

 create table SOME_DATA as
  select datepart(DATE_AUTH1) as DATE_AUTH format=mmddyy10.,PATIENT_KEY,NDC_KEY
   from connection to ORCL (
    SELECT DISTINCT DATE_AUTH1,PATIENT_KEY,NDC_KEY
     FROM DATA1
     WHERE ROWNUM<=1000
     ORDER BY 1,2,3
   );

 disconnect from ORCL;
quit;
```

Working with SAS Dates and Times


```
*-------------------------------------------------------------------------------*;
* Just for shits and giggles, determine which patients are taking more than    *;
* one drug and write those drug addicts back to the database.                   *;
*-------------------------------------------------------------------------------*;
proc means data=PATDRUGS nway noprint;
 by DATE_AUTH PATIENT_KEY;
 output out=PATDRUGS_CNT(drop=_type_ where=(_freq_>1));
run;

*-------------------------------------------------------------------------------*;
* Create a database table to hold the dates and patient keys and insert the     *;
* data back into the database table.                                            *;
*-------------------------------------------------------------------------------*;
proc sql noprint noerrorstop;
 connect to oledb as ORCL (init_string="Provider=OraOLEDB.Oracle;Data Source=ORCL;USER
ID=SCOTT;PASSWORD=TIGER;PORT=1521");

 execute(DROP TABLE DRUG_ADDICTS) by ORCL;

 execute(CREATE TABLE DRUG_ADDICTS(DATE_AUTH DATE NOT NULL,PATIENT_KEY NUMBER NOT NULL)) by
ORCL;

 insert into dbORCL.DRUG_ADDICTS(DATE_AUTH,PATIENT_KEY)
  select dhms(DATE_AUTH,0,0,0) format=datetime.,PATIENT_KEY
   from PATDRUGS_CNT;

 disconnect from ORCL;
quit;

*-------------------------------------------------------------------------------*;
* Clear the LIBNAME to the database.                                            *;
*-------------------------------------------------------------------------------*;
libname dbORCL clear;
run;
```

Working with SAS Dates and Times


```
SQL> SELECT * FROM DRUG_ADDICTS;

DATE_AUTH PATIENT_KEY
--------- -----------
28-JUL-04    26558695
04-OCT-04    26531265
17-NOV-04       22224
06-DEC-04    26441636
26-MAR-05    23749066
21-APR-05    29955020
17-MAY-05    28801004
07-JUN-05    26558572
10-OCT-05    27363582
09-NOV-05    27364557
26-DEC-05       13842
28-DEC-05    28801391
```

If you forget to use `DHMS()`, this is what you wind up with:

```
DATE_AUTH PATIENT_KEY
--------- -----------
01-JAN-60    26558695
01-JAN-60    26531265
01-JAN-60       22224
01-JAN-60    26441636
01-JAN-60    23749066
01-JAN-60    29955020
01-JAN-60    28801004
01-JAN-60    26558572
01-JAN-60    27363582
01-JAN-60    27364557
01-JAN-60       13842
01-JAN-60    28801391
```

Working with SAS Catalogs

Working with SAS Catalogs

On a day-to-day basis, SAS programmers work with SAS datasets – a special kind of *SAS File* -- which contain rows and columns of data. Many of you will be surprised to learn that there is another kind of SAS File called a *SAS Catalog*. A SAS Catalog is a special kind of SAS File that contains one or more collections of information called *catalog entries*. That is, whereas a SAS Dataset contains only one "thing" -- rows and columns of data, a SAS Catalog can contain more than one "thing".

Again, most of you will be surprised that *you create SAS Catalogs all the time* and may not know it. For instance, when you create your own SAS Format, you are creating a *SAS Format Catalog*. A SAS Format Catalog contains one or more SAS formats/informats of your own creation. When you create graphs using SAS Graph, you are creating a *SAS Graph Catalog*. A SAS Graph Catalog contains one or more SAS graphs created by the `PROC GPLOT`, etc. procedures. When you create one or more SAS Macros, you are creating a *SAS Macro Catalog*. The SAS Macro Catalog contains all of the macros you've created during your SAS session.

This section describes how to manipulate SAS Catalogs; that is, how to delete entries, rename entries, view entries, etc.

This section may not change your life, but it's worthwhile to know just in case there is an unforeseen surprise SAS Catalog quiz sometime in your future. You will become a better person for knowing this information!

Working with SAS Catalogs

Recall that a SAS Dataset can have a one-level or two-level name. The first "level" denotes where the dataset resides, in the SAS WORK area or a permanent location. The second-level name is the name of the SAS Dataset itself. For example, here I create a work dataset and a permanent dataset:

```
data mydata; /* WORK dataset – contains one level */
 a=1;output;
run;

libname SASOUT "C:\TEMP";
run;

data SASOUT.mydata; /* permanent dataset – contains two levels */
 set mydata;
run;
```

SAS Catalogs actually have 4-levels to their name:

`libname.catalog_name.catalog_entry_name.catalog_entry_type`

- The `libname` is the location where you have the SAS Catalog file (or `WORK`).
- The `catalog_name` is the name of the SAS catalog file itself. On disk, these files end in the extension *`.sas7bcat`*.
- The `catalog_entry_name` is the name of the "thing" within the SAS Catalog. For example, if you've created a SAS Format called `BOB`, then the `catalog_entry_name` would be `BOB`.
- The `catalog_entry_type` denotes the type of `catalog_entry`. The `catalog_entry_type` for our SAS Format `BOB` would be `FORMAT`.

Working with SAS Catalogs

Thus, to fully refer to our `BOB` format, the name would be something like this (assuming `LIBIN` is defined):

```
LIBIN.MYCAT.BOB.FORMAT
```

You can refer to the SAS Catalog `MYCAT` as `LIBIN.MYCAT`. Note that `MYCAT` is the `catalog_name`, `BOB` is the `catalog_entry_name`, and `FORMAT` is the `catalog_entry_type`.

Note that when you first start your SAS session, there is no Format Catalog, Graph Catalog, Macro Catalog, etc. simply because you've haven't done anything yet! But, SAS creates two catalogs by default: `SASHELP.HOST` and `SASUSER.PROFILE`. As we will see later on, once you create your own format, make a graph and create a macro, SAS will create the following SAS Catalogs: `WORK.FORMATS`, `WORK.GSEG` and `WORK.SASMACR`. Naturally, you can make these catalogs permanent; that is, `WORK.` will be replaced by `LIBOUT.`, say.

There are several `catalog_entry_type`s, similar to `FORMAT` above. Here is a brief list of them: `FORMAT`(numeric format), `FORMATC`(character format), `INFMT` (numeric informat), `INFMTC`(character informat), `REPT`(PROC REPORT definition), `TRANTAB`(translation table), `CBT`, `CLASS`, `EDPARMS`, `FRAME`, `FORM`, `HELP`, `INTRFACE`, `KEYS`, `LIST`, `MENU`, `PROGRAM`, `RANGE`, `RESOURCE`, `SCT` (SAS/AF), `MACRO`(SAS Macro), `GRSEG`(SAS Graph), etc.

Working with SAS Catalogs

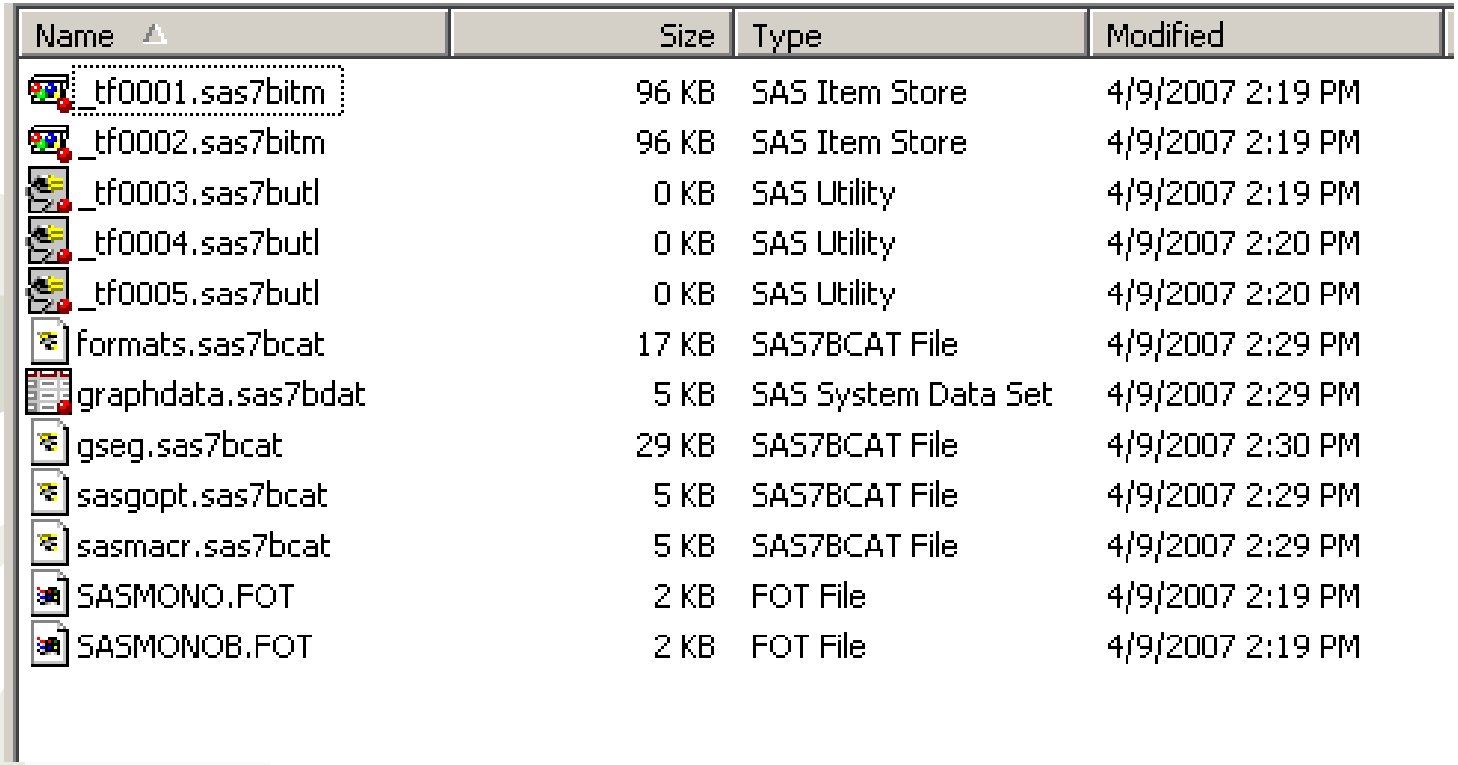
To make life interesting, let's create a SAS Format, a SAS Graph and a SAS Macro:

```
%macro DOIT;

 proc format;
  value $BOB "A"    = 1
             other = 2;
 run;

 data GraphData;
  x=1;y=1;output;
  x=2;y=2;output;
  x=3;y=3;output;
  x=4;y=4;output;
 run;

 proc gplot data=GraphData;
  plot x*y;
 run;
 quit;

%mend DOIT;
%DOIT;
```

Note that by default, all catalogs are located in the `WORK` libname.

The next slide shows the files created on disk by SAS when the macro above is run. Take note of the files ending in `.sas7bcat`. These are catalogs.

Working with SAS Catalogs



Name	Size	Type	Modified
_tf0001.sas7bitm	96 KB	SAS Item Store	4/9/2007 2:19 PM
_tf0002.sas7bitm	96 KB	SAS Item Store	4/9/2007 2:19 PM
_tf0003.sas7butl	0 KB	SAS Utility	4/9/2007 2:19 PM
_tf0004.sas7butl	0 KB	SAS Utility	4/9/2007 2:20 PM
_tf0005.sas7butl	0 KB	SAS Utility	4/9/2007 2:20 PM
formats.sas7bcat	17 KB	SAS7BCAT File	4/9/2007 2:29 PM
graphdata.sas7bdat	5 KB	SAS System Data Set	4/9/2007 2:29 PM
gseg.sas7bcat	29 KB	SAS7BCAT File	4/9/2007 2:30 PM
sasgopt.sas7bcat	5 KB	SAS7BCAT File	4/9/2007 2:29 PM
sasmacr.sas7bcat	5 KB	SAS7BCAT File	4/9/2007 2:29 PM
SASMONO.FOT	2 KB	FOT File	4/9/2007 2:19 PM
SASMONOB.FOT	2 KB	FOT File	4/9/2007 2:19 PM

Working with SAS Catalogs

Next, let's use the SAS CATALOG procedure to see the entries in each of the three SAS Catalogs:

```
proc catalog catalog=WORK.FORMATS;
 contents;
run;
quit;

proc catalog catalog=WORK.GSEG;
 contents;
run;
quit;

proc catalog catalog=WORK.SASMACR;
 contents;
run;
quit;
```

Contents of Catalog WORK.FORMATS

#	Name	Type	Create Date	Modified Date	Description
1	BOB	FORMATC	09APR2007:14:40:21	09APR2007:14:40:21	

Contents of Catalog WORK.GSEG

#	Name	Type	Create Date	Modified Date	Description
1	GPLOT	GRSEG	09APR2007:14:29:39	09APR2007:14:29:39	Plot of x * y

Contents of Catalog WORK.SASMACR

#	Name	Type	Create Date	Modified Date	Description
1	DOIT	MACRO	09APR2007:14:40:21	09APR2007:14:40:21	

Working with SAS Catalogs

Next, let's use the SAS CATALOG procedure to delete the graph GPLOT from the WORK.GSEG SAS Catalog:

```
proc catalog catalog=WORK.GSEG;
 delete GPLOT/entrytype=GRSEG;
run;
quit;
```

Note that in order to delete a graph, delete a format, or delete a macro, we have to use PROC CATALOG. We cannot use PROC DATASETS (since that's for SAS datasets, duh!), and we cannot use PROC FORMAT to delete a format and PROC GPLOT to delete a graph since these procedures do NOT contain options to allow for the deletion of a catalog entry.

The PROC CATALOG code above is very useful if you want to delete graphs as you create in order to prevent your SAS Graphics Catalog from becoming too large.

Next, let's rename the SAS Format BOB to WILMA:

```
proc catalog catalog=WORK.FORMATS;
 change BOB=WILMA/entrytype=FORMATC;
run;
quit;

Contents of Catalog WORK.FORMATS

#    Name     Type                Create Date          Modified Date    Description
-------------------------------------------------------------------------------
1    WILMA    FORMATC     09APR2007:14:40:21     09APR2007:14:40:21
```

Working with SAS Catalogs

Next, let's copy our SAS Format `WILMA` to a permanent SAS Format Catalog called `BEDROCK` located on the root of the C-Drive:

```
libname CATOUT "C:\";
run;

proc catalog;
 copy IN=WORK.FORMATS OUT=CATOUT.BEDROCK;
  select WILMA/entrytype=FORMATC;
run;
quit;

libname CATOUT clear;
run;
```

If you want to copy all formats from our `WORK.FORMATS` catalog to a permanent SAS Format Catalog called `BEDROCK`, leave off the SELECT line:

```
libname CATOUT "C:\";
run;

proc catalog;
 copy IN=WORK.FORMATS OUT=CATOUT.BEDROCK;
run;
quit;

libname CATOUT clear;
run;
```

Working with SAS Catalogs

You're all familiar with the LIBNAME and the FILENAME statements in SAS, but some of you may be unfamiliar with the CATNAME statement. The CATNAME statement allows you to *logically* concatenate SAS catalogs from disparate locations together into one large catalog during the working SAS session. Let's say that you have two SAS Format Catalogs, formats.sas7bdat, one located in the folder C:\TEMP\CAT1 and the other located in the folder C:\TEMP\CAT2. You can logically concatenate them together using the CATNAME statement like this and use PROC CATALOG against the catname:

```
libname SASIN1 "C:\TEMP\CAT1";
run;

libname SASIN2 "C:\TEMP\CAT2";
run;

proc format library=SASIN1;
 value $BOB "A"=1 other=2;
run;

proc format library=SASIN2;
 value $FRED "A"=1 other=2;
run;

catname WORK.ALLCATS ( SASIN1.formats(ACCESS=READONLY) SASIN2.formats(ACCESS=READONLY) );
run;

proc catalog catalog=WORK.ALLCATS;
 contents;
run;
quit;

catname WORK.ALLCATS clear;
run;

libname SASIN2 clear;
run;

libname SASIN1 clear;
run;
```

Using the Dictionary Tables in PROC SQL

Using the Dictionary Tables in PROC SQL

Okay, so you need to know the number of rows in your SAS dataset called `ListOfNDCKeys` because you are going to do a bunch of SAS Macro `%Do Loops` and you're gonna show the bastards!

One way of obtaining the number of rows of a SAS dataset is like this:

```
data _null_;
 set ListOfNDCKeys end=lastcase;
 if lastcase then do;
  call symput("TotNDCs",left(_N_));
 end;
run;
%put ==>&TotNDCs.<==;
```

Another way is to use `PROC CONTENTS`:

```
proc contents data=ListOfNDCKeys out=dsinfo(keep=MEMNAME NOBS) noprint;
run;

data _null_;
 set dsinfo;
 call symput("TotNDCs",left(NOBS));
run;
%put ==>&TotNDCs.<==;
```

So, which is better? Clearly the first one is not very efficient especially if the dataset `ListOfNDCKeys` is huge! The second one, while it is efficient, requires two steps: one to call the `PROC CONTENTS` and the other the create the SAS Macro variable.

Can we do better??

Using the Dictionary Tables in PROC SQL

SAS has several "views" – called Dictionary Tables – built-in to `PROC SQL` which give you access to a lot of information about SAS Datasets (like the number of rows, columns,…), SAS System Options, Formats, Macros, Libnames, etc.

Each Dictionary Table can be accessed from within `PROC SQL` by prefixing the Dictionary Table name with `DICTIONARY.` on the `FROM` clause.

Here is an example the example above again, but this time we will use the Dictionary Table `DICTIONARY.TABLES`:

```
proc sql noprint;
 select NOBS
  into :TotNDCs
  from DICTIONARY.TABLES
  where LIBNAME='WORK'
        and MEMNAME="LISTOFNDCKEYS";
quit;
%put ==>&TotNDCs.<==;
```

Thus, in one step we create the SAS Macro variable containing the number of rows of data in the SAS Dataset `ListOfNDCKeys`.

Two things to note:

1. SAS will use the `BEST12.` Format for `NOBS`, so if the number of observations is very large, best to use a `PUT()` statement and format it yourself; otherwise, SAS may turn it into scientific notation.
2. Both `LIBNAME` and `MEMNAME` expect capitalized entries.

Using the Dictionary Tables in PROC SQL

You're probably wondering how I knew DICTIONARY.TABLES contained the columns NOBS, LIBNAME and MEMNAME. Here is how you can see the layout of any Dictionary table (the output appears in the SAS Log):

```
proc sql;
 describe table DICTIONARY.TABLES;
quit;
```

…produces…

```
create table DICTIONARY.TABLES
  (
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   dbms_memtype char(32) label='DBMS Member Type',
   memlabel char(256) label='Dataset Label',
   typemem char(8) label='Dataset Type',
   crdate num format=DATETIME informat=DATETIME label='Date Created',
   modate num format=DATETIME informat=DATETIME label='Date Modified',
   nobs num label='Number of Physical Observations',
   obslen num label='Observation Length',
   nvar num label='Number of Variables',
   protect char(3) label='Type of Password Protection',
   compress char(8) label='Compression Routine',
   encrypt char(8) label='Encryption',
   npage num label='Number of Pages',
   filesize num label='Size of File',
   pcompress num label='Percent Compression',
   reuse char(3) label='Reuse Space',
   bufsize num label='Bufsize',
   delobs num label='Number of Deleted Observations',
   nlobs num label='Number of Logical Observations',
```

Using the Dictionary Tables in PROC SQL


```
    maxvar num label='Longest variable name',
    maxlabel num label='Longest label',
    maxgen num label='Maximum number of generations',
    gen num label='Generation number',
    attr char(3) label='Dataset Attributes',
    indxtype char(9) label='Type of Indexes',
    datarep char(32) label='Data Representation',
    sortname char(8) label='Name of Collating Sequence',
    sorttype char(4) label='Sorting Type',
    sortchar char(8) label='Charset Sorted By',
    reqvector char(24) format=$HEX48 informat=$HEX48 label='Requirements Vector',
    datarepname char(170) label='Data Representation Name',
    encoding char(256) label='Data Encoding',
    audit char(3) label='Audit Trail Active?',
    audit_before char(3) label='Audit Before Image?',
    audit_admin char(3) label='Audit Admin Image?',
    audit_error char(3) label='Audit Error Image?',
    audit_data char(3) label='Audit Data Image?'
  );
```

Take note of following columns: `DELOBS` and `NLOBS`. When using PROC SQL, the deletion of rows can be marked in the dataset, but the observation can actually still be in the dataset itself. `DELOBS` tells you the number of deleted observations, whereas `NLOBS` gives you the number of observations less the number of deleted observations. Thus, the SELECT line should be either this:

```
select NOBS-DELOBS
```

or this

```
select NLOBS
```

Using the Dictionary Tables in PROC SQL

Here is a list of all of the Dictionary Tables, and the definition of a few of them:

CATALOGS: Contains information about known SAS catalogs.
CHECK_CONSTRAINTS: Contains information about known check constraints.
COLUMNS: Contains information about columns in all known tables.
CONSTRAINT_COLUMN_USAGE: Contains information about columns that are referred to by integrity constraints.
CONSTRAINT_TABLE_USAGE: Contains information about tables that have integrity constraints defined on them.
DICTIONARIES: Contains information about all DICTIONARY tables.
EXTFILES: Contains information about known external files.
FORMATS: Contains information about currently accessible formats and informats.
GOPTIONS: Contains information about currently defined graphics options (SAS/GRAPH software).
INDEXES: Contains information about known indexes.
LIBNAMES: Contains information about currently defined SAS data libraries.
MACROS: Contains information about currently defined macros.
MEMBERS: Contains information about all objects that are in currently defined SAS data libraries.
OPTION: Contains information on SAS system options.
REFERENTIAL_CONSTRAINTS: Contains information about referential constraints.
STYLES: Contains information about known ODS styles.
TABLE_CONSTRAINTS: Contains information about integrity constraints in all known tables.
TABLES: Contains information about known tables.
TITLES: Contains information about currently defined titles and footnotes.
VIEWS: Contains information about known data views.

```
create table DICTIONARY.OPTIONS
  (
   optname char(32) label='Option Name',
   opttype char(8) label='Option type',
   setting char(1024) label='Option Setting',
   optdesc char(160) label='Option Description',
   level char(8) label='Option Location',
   group char(32) label='Option Group'
  );
```

Using the Dictionary Tables in PROC SQL


```
create table DICTIONARY.LIBNAMES
  (
   libname char(8) label='Library Name',
   engine char(8) label='Engine Name',
   path char(1024) label='Path Name',
   level num label='Library Concatenation Level',
   fileformat char(8) label='Default File Format',
   readonly char(3) label='Read-only?',
   sequential char(3) label='Sequential?',
   sysdesc char(1024) label='System Information Description',
   sysname char(1024) label='System Information Name',
   sysvalue char(1024) label='System Information Value'
  );

create table DICTIONARY.COLUMNS
  (
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   name char(32) label='Column Name',
   type char(4) label='Column Type',
   length num label='Column Length',
   npos num label='Column Position',
   varnum num label='Column Number in Table',
   label char(256) label='Column Label',
   format char(49) label='Column Format',
   informat char(49) label='Column Informat',
   idxusage char(9) label='Column Index Type',
   sortedby num label='Order in Key Sequence',
   xtype char(12) label='Extended Type',
   notnull char(3) label='Not NULL?',
   precision num label='Precision',
   scale num label='Scale',
   transcode char(3) label='Transcoded?'
  );
```

Using the Dictionary Tables in PROC SQL

Let's see an example of how to use DICTIONARY.COLUMNS to get a list of variables for a particular dataset:

```
data test1;
 var1=1;var2='A';var3=17;var4='Bob';output;
run;

proc sql;
 select libname,memname,name,type,length,varnum
  from DICTIONARY.COLUMNS
  where libname='WORK'
        and memname='TEST1';
quit;
```

Library Name	Member Name	Column Name	Column Type	Column Length	Column Number in Table
WORK	TEST1	var1	num	8	1
WORK	TEST1	var2	char	1	2
WORK	TEST1	var3	num	8	3
WORK	TEST1	var4	char	3	4

To create macro variables from the variable names:

```
proc sql;
 select name
  into :v1-:v4
  from DICTIONARY.COLUMNS
  where libname='WORK'
        and memname='TEST1';
quit;
/* v1=var1, v2=var2, v3=var3, v4=var4 */
```

```
proc sql;
 select name
  into :varz separated by ","
  from DICTIONARY.COLUMNS
  where libname='WORK'
        and memname='TEST1';
quit;
/* varz=var1,var2,var3,var4 */
```

Creating a SAS Transport File

Creating a SAS Transport File

A *SAS Transport file* is a SAS dataset put into a generic file format that can be sent to a client who's SAS version and/or operating system is unknown ahead of time. In many cases, especially with older versions of SAS, you cannot just send a SAS dataset via email or FTP. Although creating SAS transport files does not come up too often, being aware of how to create a SAS transport file is a good thing.

There are several ways to create a SAS transport file:

1. Use the `XPORT` engine along with either a DATA Step or `COPY` procedure
2. Use `PROC CPORT` to create a transport file and use `PROC CIMPORT` to read in the transport file to reconstitute the SAS dataset
3. Use Cross-Environment Data Access (`CEDA`).
4. Use the XML Libname (see section on XML Libname later in the presentation)

The `XPORT` engine only works with SAS Datasets and does not allow you to create a transport file for SAS Catalogs. Also, any long variable names are truncated to 8 characters (must specify the `VALIDVARNAME=V6` SAS System Option).

`PROC CPORT`/`PROC CIMPORT` work with both SAS Datasets and SAS Catalogs. But, these procedures do NOT allow you to *regress* the datasets/catalogs; that is, you cannot send a transport file containing a SAS Dataset/Catalog created in SAS Version 9 to a client running SAS Version 6.

Cross-Environment Data Access (`CEDA`) is limited to SAS Versions 8 and 9, and only works on SAS Datasets, not SAS Catalogs. `CEDA` does NOT require that you create a transport file, but just hand the SAS dataset to the anxious user. The SAS Dataset will be converted without the user's knowledge.

Creating a SAS Transport File

Creating a Transport File from a SAS Dataset Using the XPORT Engine

```
options validvarname=v7;
run;

data test1;
 mylongvariablename=1;output;
run;

/* Needed or PROC COPY will fail! */
options validvarname=v6;
run;

/* Note that XFILE.XPT is a physical file and not just a library */
libname xfile xport "C:\TEMP\xfile.xpt";
run;

proc copy in=work out=xfile memtype=data;
 select test1;
run;
quit;

libname xfile clear;
run;
```

Creating a SAS Transport File

Creating a Transport File From a Library Containing Multiple SAS Datasets Using the XPORT Engine

```
options validvarname=v7;
run;

data test1;
 mylongvariablename=1;output;
run;

data test2;
 mylongvariablename=1;output;
run;

/* Needed or PROC COPY will fail! */
options validvarname=v6;
run;

/* Note that XFILE.XPT is a physical file and not just a library */
libname xfile xport "C:\TEMP\xfile.xpt";
run;

proc copy in=work out=xfile memtype=data;
run;
quit;

libname xfile clear;
run;
```

Creating a SAS Transport File

sheepsqueezers.com

Creating a Transport File From a SAS Dataset Using the PROC CPORT Engine

```
filename cportout "c:\cptfile.xpt";
run;

/* Can use SELECT statement with PROC CPORT! */
proc cport data=work.test1 file=cportout;
run;

filename cportout clear;
run;
```

Creating a Transport File From a SAS Library Using the PROC CPORT Engine

```
filename cportout "c:\cptfile.xpt";
run;

proc cport library=work file=cportout;
run;

filename cportout clear;
run;
```

Creating a SAS Transport File

Using the `PROC CIMPORT` Procedure to Reconstitute the SAS Datasets

```
filename cportin "c:\cptfile.xpt";
run;

proc cimport infile=cportin library=work;
run;

filename cportin clear;
run;
```

Working with SAS Formats

Working with SAS Formats

In previous sections of this course, we've seen the uses of the SAS procedure `PROC FORMAT` in creating our own formats and informats as well as the usefulness of the built-in formats and informats provided by SAS Institute with the SAS System (such as `mmddyy10.`, `datetime.`, etc.).

In this section, we will introduce the SAS procedure `PROC FORMAT` more fully including a discussion on how to create your own formats and informats *on-the-fly* as opposed to coding `PROC FORMAT` statements by hand.

We will also introduce the SAS functions `PUT()`, `INPUT()`, `PUTC()`, `PUTN()`, `INPUTC()` and `INPUTN()` which use SAS formats and informats within a SAS Data Step.

The SAS Documentation uses the terminology *the informat converts*, *the format prints*. In general, *informats* are used to convert numbers or characters to other numbers or characters, whereas formats are generally used for *pretty* printing such as changing a number to a character string, or a character string to another character string.

In general, you use ***in***formats with the ***`IN`***`PUT*()` functions, or on the ***`IN`***`PUT` statement while reading in a text file. You use formats with the `PUT*()` functions, on the `PUT` statement while writing to a text file and the `FORMAT` statement. Both can be used with the `ATTRIB` statement in the Data Step.

Finally, you can use your own *homemade* formats and informats wherever you can use SAS-supplied formats and informats.

Working with SAS Formats

Here is an example I call the "You already know this!" example:

```
/* Use INFORMATs below to read in the data */
data EmpData;
 infile cards;
 input @1  Name          $char9.0
       @12 EmployeeID         5.0
       @20 YearlySalary comma9.2;
cards;
Bob Smith  12345   85,123.45
Moe Smith  12346   95,123.45
;
run;

/* Use FORMATs below to print out or write data to a file */
proc print data=EmpData;
 var EmployeeID Name YearlySalary;
 format YearlySalary comma9.2;
run;

/* Use FORMATs below */
data _null;
 file "C:\TEMP\MyFlatFile.txt" notitles noprint;
 set EmpData;
 put @1  Name           $char10.0
     @20 EmployeeID          10.0
     @40 YearlySalary  comma12.2;
run;
```

Working with SAS Formats

SAS Formats

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.

In the following example, we create a **SAS Character Format**. This Character Format converts `NDC_KEYs` to label names:

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

Take note of the following things:

1. Both Character *Formats* and Numeric *Formats* use the VALUE keyword
2. A Character Format name starts with a dollar sign ($): $NDC
3. You can specify multiple formats/informats in one PROC FORMAT procedure, just be sure to end each one with a semicolon.
4. Note the each NDC_KEY is specified on the left side, followed by an equal sign, followed by the resulting target character string.

Working with SAS Formats

SAS Formats (continued)

Here is an example of how to use this format:

```
/* Character Format */
proc format;
 value $NDC "12345678901" = "SAMPLEX 10MG BOTTLE 100"
            "12345678902" = "SAMPLEX 20MG BOTTLE 100"
            "12345678903" = "SAMPLEX 30MG BOTTLE 100"
            "12345678904" = "SAMPLEX 40MG BOTTLE 100"
 ;
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	SAMPLEX 10MG BOTTLE 100
2	99999999999	99999999999

1. Since the NDC_KEY="99999999999" is not defined in the character format $NDC, SAS just returns the original data.
2. Take note that the character format $NDC is postfixed with a period above.

Working with SAS Formats

SAS Formats (continued)

Let's say that we want any NDC_KEY not specifically defined in our character format to be given the text UNKNOWN LABEL NAME. Here's how to do that:

```
/* Character Format */
proc format;
 value $NDC "12345678901" = "SAMPLEX 10MG BOTTLE 100"
            "12345678902" = "SAMPLEX 20MG BOTTLE 100"
            "12345678903" = "SAMPLEX 30MG BOTTLE 100"
            "12345678904" = "SAMPLEX 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	SAMPLEX 10MG BOTTLE 100
2	99999999999	UNKNOWN LABEL NAME

Working with SAS Formats

SAS Formats

In the following example, we create a **SAS Numeric Format**. This Numeric Format converts ranges of AGE_KEYs to textual descriptions:

```
/* 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;
```

Take note of the following things:

1. Both Character *Formats* and Numeric *Formats* use the VALUE keyword
2. A Numeric Format names do **NOT** start with a dollar sign ($): AGE
3. You can specify multiple formats/informats in one `PROC FORMAT` procedure, just be sure to end each one with a semicolon.
4. Note the each AGE_KEY is specified on the left side in this example as a range of ages with a dash between each starting and ending age.
5. Take note of the `high` keyword. There is also a `low` keyword as well.

Working with SAS Formats

SAS Formats (continued)

Here is an example of how to use this format:

```
/* 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

Working with SAS Formats

SAS Formats (continued)

You may be asking what would happen if an AGE_KEY was, say, 12.5. In this case, zero to 12 is defined, and 13 to 18 is defined, but nothing between 12 and 13 is defined. Here is how to rectify that:

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

data MyAgeData;
 AGE_KEY=12;AgeDesc=put(AGE_KEY,AGE.);output;
 AGE_KEY=12.5;AgeDesc=put(AGE_KEY,AGE.);output;
 AGE_KEY=13;AgeDesc=put(AGE_KEY,AGE.);output;
run;

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

Obs	AGE_KEY	AgeDesc
1	12.0	Prepubescent
2	**12.5**	**Prepubescent**
3	13.0	Just Trouble

Working with SAS Formats

SAS Formats (continued)

Here is another example of how to use this format:

```
proc print data=MyAgeData;
 var AGE_KEY;
 format AGE_KEY AGE.;
run;
```

Some Comments About Character and Numeric Formats:

1. Ensure that the text to the right of the equal sign is enclosed in quotes and all have the same length (as in the example on the previous slide). This will prevent SAS from assigning a character length to your variable that is shorter than the largest target text
2. Best to line up the equal signs…could indicate a problem if there is misalignment
3. Ensure that you have placed a dollar sign before the format name for character formats, but **not** for numeric formats
4. Ensure that you have placed a period after the character or numeric format name when being used in the `PUT()` function on the `FORMAT` statement
5. Use the `OTHER` keyword to capture any data that has not be assigned
6. To ensure your ranges don't include the endpoint, use the less than symbol.
7. Note that the `PUT()` function **ALWAYS** returns a character variable!!
8. Character Formats take a dollar-sign before the format name whereas Numeric Formats do not.

Working with SAS Formats

SAS Formats (continued)

You can specify formats for your variables in a SAS Data Step by using the ATTRIB or FORMAT statements:

```
data AgeAndNDCData;
 ATTRIB AGE_KEY format=AGE.
        NDC_KEY format=$NDC.;
 /*
  FORMAT AGE_KEY AGE. NDC_KEY $NDC.;
 */
 AGE_KEY=6;NDC_KEY="12345678901";output;
 AGE_KEY=15;NDC_KEY="12345678902";output;
run;

proc print data=AgeAndNDCData;
 var AGE_KEY NDC_KEY;
run;
```

Obs	AGE_KEY	NDC_KEY
1	Prepubescent	SAMPLEX 10MG BOTTLE 100
2	Just Trouble	SAMPLEX 20MG BOTTLE 100

Here is a partial PROC CONTENTS listing showing the associated formats:

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format
1	AGE_KEY	Num	8	**AGE.**
2	NDC_KEY	Char	24	**$NDC.**

Working with SAS Formats

SAS Formats (continued)

It may be just crazy, but sometimes you may want to disassociate a format from a variable. Here is how to do this:

```
data AgeAndNDCData;
 set AgeAndNDCData;
 FORMAT AGE_KEY NDC_KEY;
run;
```

Make sure that the `FORMAT` line is **AFTER** the `SET` statement; otherwise, your formats won't be removed!! Yikes!!

Working with SAS Formats

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;
```

You'll notice the following things:

1. The keyword `INVALUE` is being used
2. The Numeric Informat name `AGEGRP` is NOT preceded with a dollar-sign
3. You can use the less-than symbol with SAS Informats
4. Can place comments in `PROC FORMAT`

Working with SAS Formats

SAS Informats (continued)

SAS Numeric Informat – Convert Number to Number (continued)

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

```
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.);
 output;
 AGE_KEY=66;
 AgeGroup=input(AGE_KEY,AGEGRP.);
 output;
run;
```

Obs	AGE_KEY	AgeGroup
1	12	1
2	25	3
3	66	4

Working with SAS Formats

SAS Informats (continued)

SAS Numeric Informat – Convert Character to Number

In the following example, we create a SAS Numeric Informat which converts one set of characters (NDC_KEYs) to a set of numbers (PROD_GROUP).

```
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;
```

You'll notice the following things:

1. The keyword INVALUE is being used
2. The Numeric Informat name PGRP is NOT preceded with a dollar-sign
3. Can place comments in PROC FORMAT

Working with SAS Formats

SAS Informats (continued)

SAS Numeric Informat – Convert Character to Number (continued)

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

Working with SAS Formats

SAS Informats (continued)

SAS Character Informat – Convert Character to Character

In the following example, we create a SAS Character Informat which converts one set of characters (`ZIP`) to another set of characters (`CitySt`).

```
proc format;
 invalue $Z2S "19115" = "Philadelphia,PA    "
              "19042" = "Plymouth Meeting,PA"
              other   = "The Boonies        "
 ;
run;
```

You'll notice the following things:

1. The keyword `INVALUE` is being used
2. The Numeric Informat name `Z2S` is preceded with a dollar-sign

Working with SAS Formats

SAS Informats (continued)

SAS Character Informat – Convert Character to Character (continued)

Here is how you use the SAS Character Informat `Z2S`:

```
proc format;
 invalue $Z2S "19115" = "Philadelphia,PA    "
              "19042" = "Plymouth Meeting,PA"
              other   = "The Boonies         "
 ;
run;

data MyZipData;
 ZIP="19115";CitySt=input(ZIP,$Z2S.);output;
 ZIP="19042";CitySt=input(ZIP,$Z2S.);output;
 ZIP="99999";CitySt=input(ZIP,$Z2S.);output;
run;
```

Obs	ZIP	CitySt
1	19115	Philadelphia,PA
2	19042	Plymouth Meeting,PA
3	99999	The Boonies

Working with SAS Formats

SAS Informats (continued)

SAS Character Informat – Convert Number to Character

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

```
proc format;
 invalue $AGRP   0-12   = "Prepubescent            "
                13-19   = "Just Trouble            "
                20-63   = "Working Stiff           "
                64-high = "Knocking on Death`s Door"
                other   = "Unknown Age             "
 ;
run;
```

You'll notice the following things:

1. The keyword `INVALUE` is being used
2. The Character Informat name `AGRP` is preceded with a dollar-sign

Working with SAS Formats

SAS Informats (continued)

SAS Character Informat – Convert Number to Character (continued)

Here is how you use the SAS Character Informat `$AGRP`:

```
proc format;
 invalue $AGRP   0-12   = "Prepubescent            "
                13-19   = "Just Trouble            "
                20-63   = "Working Stiff           "
                64-high = "Knocking on Death`s Door"
                other   = "Unknown Age             "
 ;
run;

data MyAgeData2;
 length AgeGroup $ 25;
 AGE_KEY=1;
 AgeGroup=input(trim(left(put(AGE_KEY,2.))),$AGRP.);
 output;
 AGE_KEY=12;
 AgeGroup=input(trim(left(put(AGE_KEY,2.))),$AGRP.);
 output;
 AGE_KEY=25;
 AgeGroup=input(trim(left(put(AGE_KEY,2.))),$AGRP.);
 output;
 AGE_KEY=66;
 AgeGroup=input(trim(left(put(AGE_KEY,2.))),$AGRP.);
 output;
run;
```

Obs	AgeGroup	AGE_KEY
1	Prepubescent	1
2	Prepubescent	12
3	Working Stiff	25
4	Knocking on Death`s Door	66

Working with SAS Formats

SAS Informats (continued)

Here is another example of how to use a Character Informat, only this time on an `INPUT` statement and not with an `INPUT()` Function:

```
proc format;
 invalue $SEX    "M"    = "Male    "
                 "F"    = "Female  "
                 other  = "Unknown"
 ;
run;

data MySexData;
 infile cards;
 input @1 GenderDesc $SEX.;
cards;
M
F
U
;
run;
```

Obs	GenderDesc
1	Male
2	Female
3	Unknown

Working with SAS Formats

SAS Formats and Informats – Additional Features

There are several additional features you can use with `PROC FORMAT` when creating SAS Formats and Informats.

You can use the keyword `_SAME_` to indicate that the SAS Informat should not change the raw data in any way:

```
proc format;
 invalue $NCHK "12345678901",
               "12345678902",
               "12345678903",
               "12345678904" = _same_
               other         = "XXXXXXXXXXX"
 ;
run;

data NDC_Check;
 NDC_KEY="98765432109";Check=input(NDC_KEY,$NCHK.);output;
 NDC_KEY="12345678901";Check=input(NDC_KEY,$NCHK.);output;
run;

Obs      NDC_KEY          Check
 1     98765432109     XXXXXXXXXXX
 2     12345678901     12345678901
```

Working with SAS Formats

SAS Formats and Informats – Additional Features

You can use the keyword _ERROR_ to indicate that the SAS Informat should assign a missing value to that data and issue a warning message in the SAS Log:

```
proc format;
 invalue $NCHK "12345678901",
               "12345678902",
               "12345678903",
               "12345678904" = _same_
               other         = _error_
 ;
run;

data NDC_Check;
 NDC_KEY="98765432109";Check=input(NDC_KEY,$NCHK.);output;
 NDC_KEY="12345678901";Check=input(NDC_KEY,$NCHK.);output;
run;
```

NOTE: Invalid argument to function INPUT at line 801 column 30.
NDC_KEY=12345678901 Check=12345678901 _ERROR_=1 _N_=1

Working with SAS Formats

SAS Formats and Informats – Additional Features

You can use a SAS-supplied format or informat within your own `PROC FORMAT` by specifying the format/informat in brackets:

```
proc format;
 value DTCONV      0-50000 = [mmddyy10.]
                50001-high = [dateampm19.]
                other      = .
 ;
run;

data Dates_OR_DateTimes;
 Date_OR_DateTime=today();output;
 Date_OR_DateTime=datetime();output;
run;

proc print data=Dates_OR_DateTimes;
 var Date_OR_DateTime;
 format Date_OR_DateTime DTCONV.;
run;
```

JUST AN EXAMPLE!!
DON'T USE THIS! ☺

Obs	Date_OR_DateTime
1	04/12/2007
2	12APR07:03:23:29 PM

Working with SAS Formats

SAS Formats and Informats – Additional Features

As we talked about in the section on how to work with SAS Catalogs, PROC FORMAT creates a SAS Catalog which contains both your user-created formats and informats. This format catalog is only temporary and is located in the SAS Work directory for the duration of your SAS session. If you would like to make your formats and informats permanent so that you can use them at a later time, you must specify a place for SAS to save your format catalog. Here is an example:

```
libname MyFmtLib "C:\Temp\Formats";
run;

/* MyFmtLib is the library whereas CrappyFmts is the name of the Format Catalog */
/* You can leave off .CrappyFmts and SAS will create a catalog named FORMATS */
proc format library=MyFmtLib.CrappyFmts;
 invalue $SEX    "M"    = "Male    "
                 "F"    = "Female  "
                 other  = "Unknown"
 ;
 value $NDC "12345678901"="SAMPLEX 10MG BOTTLE 100"
            "12345678902"="SAMPLEX 20MG BOTTLE 100"
            "12345678903"="SAMPLEX 30MG BOTTLE 100"
            "12345678904"="SAMPLEX 40MG BOTTLE 100"
            other         = "OTHER LABEL NAME"
 ;
run;

libname MyFmtLib clear;
run;
```

Working with SAS Formats

SAS Formats and Informats – Additional Features

In order to use your saved formats, you must tell SAS where to look for your formats. Here is an example:

```
libname MyFmtLib "C:\TEMP\Formats";
run;

/* Note that if you let SAS name the catalog FORMATS, you do not need .CrappyFmts */
options fmtsearch=(work library MyFmtLib.CrappyFmts);
run;

data TestSex;
 GenderDesc=input("M",$sex.);output;
run;

libname MyFmtLib clear;
run;
```

Naturally, you can use `PROC CATALOG` to copy your format catalog from the `WORK` folder to a permanent folder:

```
libname MyFmtLib "C:\TEMP\Formats";
run;

proc catalog catalog=WORK.FORMATS;
 copy out=MyFmtLib.CrappyFmts2;
run;
quit;
```

Working with SAS Formats

SAS Formats and Informats – Additional Features

Occasionally, you want to see the formats that are available in the formats catalog (whether in the `WORK` directory or in a more permanent directory). You can use the `FMTLIB` or `PAGE` options to show the available formats. The `FMTLIB` option tries to put as much format information on a page, whereas the `PAGE` option places one format per page.

```
proc format fmtlib;
run;
```

```
----------------------------------------------------------------------------
|       FORMAT NAME: DTCONV   LENGTH:    19   NUMBER OF VALUES:    3       |
|   MIN LENGTH:   1  MAX LENGTH:  40  DEFAULT LENGTH  19  FUZZ: STD        |
|--------------------------------------------------------------------------|
|START           |END             |LABEL   (VER. V7|V8   12APR2007:16:51:08)|
|----------------+----------------+-----------------------------------------|
|               0|           50000|[MMDDYY10.]                              |
|          500001|HIGH            |[DATEAMPM19.]                            |
|**OTHER**       |**OTHER**       |.                                        |
----------------------------------------------------------------------------
```

Note that you can select for a specify format/informat to display:

```
proc format fmtlib;
 select DTCONV;
run;
```

Note: Precede any character or numeric **informat** name with an at-sign (`@` or `@$`).

Working with SAS Formats

Using CNTLIN and CNTLOUT

As we've seen on the previous slides, we can create SAS Formats and Informats *digitally*; that is, by hand. SAS Institute, being of sound mind and body, have written `PROC FORMAT` so that you can create SAS Formats/Informats on-the-fly by creating a SAS Dataset with the appropriate variables and specify that dataset name in the `PROC FORMAT CNTLIN=`*`dataset`* option. For example, suppose you have a SAS dataset of `NDC_KEY`s and associated label names (`LN`s). You can create a SAS Character Format *on-the-fly* to map each `NDC_KEY` to its associated `LN` by specifying the format name on the `FORMAT` statement, `ATTRIB` statement, `PUT` line, or in the `PUT()` function. Yes, you could code this by hand as in the examples above, but you'll probably go blind after about the 100th `NDC_KEY`.

You may want to talk to the other SAS programmers in your company. It is very possible that he/she has created a huge list of SAS formats specifically for use with your data. If so, this will save you time since you won't have to re-create the formats. On the other hand, if no one has created formats…maybe you should be the one to suggest it!

Not only can you create formats on-the-fly using `CNTLIN=`, but you can "unload" a SAS Format/Informat into a SAS dataset so that you can use it for your own processing by using the `PROC FORMAT CNTLOUT=`*`dataset`* option.

It may not surprise you to know that the layout of the SAS dataset created from the `CNTLOUT=` option, and the layout of the SAS dataset used by the `CNTLIN=` option are exactly the same (woohoo!). We start by describing `CNTLOUT=`.

Working with SAS Formats

Let's assume that we have created this character format:

```
proc format;
 value $NDC "12345678901"="SAMPLEX 10MG BOTTLE 100"
            "12345678902"="SAMPLEX 20MG BOTTLE 100"
            "12345678903"="SAMPLEX 30MG BOTTLE 100"
            "12345678904"="SAMPLEX 40MG BOTTLE 100"
            other        = "OTHER LABEL NAME       "
 ;
run;

proc format cntlout=fmt_ndc(keep=FMTNAME START END LABEL TYPE HLO);
 select $NDC;
run;
```

FMTNAME	START	END	LABEL	TYPE	HLO
NDC	12345678901	12345678901	SAMPLEX 10MG BOTTLE 100	C	
NDC	12345678902	12345678902	SAMPLEX 20MG BOTTLE 100	C	
NDC	12345678903	12345678903	SAMPLEX 30MG BOTTLE 100	C	
NDC	12345678904	12345678904	SAMPLEX 40MG BOTTLE 100	C	
NDC	**OTHER**	**OTHER**	OTHER LABEL NAME	C	O

FMTNAME= the name of the SAS Format/Informat
START = Staring Value of range
END = Ending Value of range
LABEL = Result of applying the format
TYPE = Specifies that this format is a Character Format (C)
HLO = Specifies that the current row in the CNTLOUT dataset refers to OTHER=. The Letter "O" indicates other while blank indicates **not** other rows.

Working with SAS Formats

Let's briefly discuss the TYPE variable. In order for PROC FORMAT to know what type of format you are creating – character informat, numeric informat, character format, or numeric format – you have to specify one of the following for the TYPE variable:

C = Character Format N = Numeric Format

J = Character Informat I = Numeric Informat

Let's briefly discuss the HLO variable. In order for PROC FORMAT to indicate that a particular row is associated with "OTHER=", the variable HLO will be set to the capital letter "O"; otherwise, leave it blank. If you specify "H" instead of "O", then PROC FORMAT knows that the row is a "-HIGH" row. If you specify "L" instead of "O", then PROC FORMAT knows that the row is a "LOW-" row. For example,

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

proc format cntlout=fmt_age(keep=FMTNAME START END LABEL TYPE HLO);
 select AGE;
run;
```

Obs	FMTNAME	START	END	LABEL	TYPE	HLO
1	AGE	LOW	12	Prepubescent	N	**L**
2	AGE	13	18	Just Trouble	N	
3	AGE	19	62	Working Stiff	N	
4	AGE	63	HIGH	Knocking on Death`s Door	N	**H**
5	AGE	**OTHER**	**OTHER**	Unknown Age	N	**O**

Working with SAS Formats

Let's briefly discuss the START and END variables. In order to define the starting range and the ending range of the format/informat (for example, our AGE format), you need to fill in the START and END variables. START is the starting value of the range whereas END is the ending value of the range. Note that both of these variables are character variables (which may seem strange for numeric formats).

Let's briefly discuss the LABEL variable. You fill in the LABEL variable with the data that would be on the right side of the equal sign if you were hand-coding the PROC FORMAT. This variable can also contain the name of a pre-existing SAS Format or Informat (recall the DTCONV example above) **without** the left and right brackets!

Let's briefly discuss the FMTNAME variable. You fill in the FMTNAME variable with the name of the format/informat you are trying to create. Note that you do **NOT** need to put in the dollar-sign and you do **NOT** need to put in the period. SAS format names can be about 30 characters long, but stick to only a few characters to save keystrokes. Note that in the previous examples, the FMTNAME is the **SAME** for each row of CNTLOUT= data.

On the following slides, we show how to use CNTLIN= to create SAS character formats, numeric formats, character informats and numeric informats on-the-fly. Take note that the CNTLIN= SAS dataset **CAN** contain several formats/informats in the one dataset! Just be sure to specify the FMTNAME and TYPE properly!!

Working with SAS Formats

Using CNTLIN= to Create a Character Format

Let's assume we have a SAS dataset containing NDC_KEYs and LNs:

```
data NDC_AND_LN;
 length NDC_KEY $ 11 LN $ 30;
 NDC_KEY="12345678901"; LN="SAMPLEX 10MG BOTTLE 100";output;
 NDC_KEY="12345678902"; LN="SAMPLEX 20MG BOTTLE 100";output;
 NDC_KEY="12345678903"; LN="SAMPLEX 30MG BOTTLE 100";output;
 NDC_KEY="12345678904"; LN="SAMPLEX 40MG BOTTLE 100";output;
run;
```

Let's create a CNTLIN dataset mapping the NDC_KEY to the LN:

```
data fmt(keep=fmtname hlo type start end label);
 length fmtname $ 8 hlo $ 1 type $ 1 start end $ 11 label $ 30;
 retain fmtname 'N2L' hlo ' ' type 'C';
 set NDC_AND_LN end=lastcase;
 start=NDC_KEY;
 end=NDC_KEY;
 label=LN;
 output;
 if lastcase then do;
  hlo='O';
  label='** NO LABEL NAME SPECIFIED **';
  output;
 end;
run;

proc format cntlin=fmt;
run;
```

Working with SAS Formats

Using CNTLIN= to Create a Character Format (continued)

Let's use the format now:

```
data NDCData;
 length NDC_KEY $ 11 LN $ 30;
 NDC_KEY="12345678901"; LN=put(NDC_KEY,$N2L.); output;
 NDC_KEY="99999999999"; LN=put(NDC_KEY,$N2L.); output;
run;
```

Obs	NDC_KEY	LN
1	12345678901	SAMPLEX 10MG BOTTLE 100
2	99999999999	** NO LABEL NAME SPECIFIED **

Note that instead of actually creating the `LN` variable, you could just use the `$N2L` format in the `PROC PRINT`:

```
proc print data=NDCData;
 var NDC_KEY;
 format NDC_KEY $N2L.;
run;
```

Working with SAS Formats

Using CNTLIN= to Create a Numeric Format

Let's assume we have a SAS dataset containing the following data:

```
data AGE_MAP;
 length START_AGE_RANGE END_AGE_RANGE 8 AGEGROUP $ 25;
 START_AGE_RANGE=0; END_AGE_RANGE=13; AGEGROUP="Prepubescent";output;
 START_AGE_RANGE=14; END_AGE_RANGE=19; AGEGROUP="Just Trouble";output;
 START_AGE_RANGE=20; END_AGE_RANGE=63; AGEGROUP="Working Stiff";output;
 START_AGE_RANGE=64; END_AGE_RANGE=99; AGEGROUP="Knocking on Death`s Door";output;
run;
```

Let's create a `CNTLIN` dataset mapping ages to groups:

```
data fmt(keep=fmtname hlo type start end label);
 length fmtname $ 8 hlo $ 1 type $ 1 start end 8 label $ 25;
 retain fmtname 'A2G' hlo ' ' type 'N';
 set AGE_MAP end=lastcase;
 start=START_AGE_RANGE;
 end=END_AGE_RANGE;
 label=AGEGROUP;
 output;
 if lastcase then do;
  hlo='O';
  label='** UNKNOWN AGE GROUP **';
  output;
 end;
run;

proc format cntlin=fmt;
run;
```

Working with SAS Formats

Using CNTLIN= to Create a Numeric Format (continued)

Let's use the format now:

```
data AgeGroups;
 AGE_KEY=14;AgeGroup=put(AGE_KEY,A2G.);output;
 AGE_KEY=29;AgeGroup=put(AGE_KEY,A2G.);output;
 AGE_KEY=76;AgeGroup=put(AGE_KEY,A2G.);output;
 AGE_KEY=-1;AgeGroup=put(AGE_KEY,A2G.);output;
run;
```

Obs	AGE_KEY	AgeGroup
1	14	Just Trouble
2	29	Working Stiff
3	76	Knocking on Death`s Door
4	-1	** UNKNOWN AGE GROUP **

Working with SAS Formats

Using CNTLIN= to Create a Numeric Informat

Let's assume we have a SAS dataset containing the following data:

```
data AGEGRPNUM;
 length START_AGE_RANGE END_AGE_RANGE 3 AGEGROUP 3;
 START_AGE_RANGE= 0; END_AGE_RANGE=13; AGEGROUP=1; output;
 START_AGE_RANGE=13; END_AGE_RANGE=19; AGEGROUP=2; output;
 START_AGE_RANGE=19; END_AGE_RANGE=63; AGEGROUP=3; output;
 START_AGE_RANGE=63; END_AGE_RANGE=99; AGEGROUP=4; output;
run;
```

Let's create a `CNTLIN` dataset mapping ages to groups:

```
data fmt(keep=fmtname hlo type start end label sexcl eexcl);
 length fmtname $ 8 hlo $ 1 type $ 1 sexcl eexcl $ 1 start end label 8;
 retain fmtname 'A2GN' hlo 'I' type 'I' sexcl 'N';
 set AGEGRPNUM end=lastcase;
 start=START_AGE_RANGE;
 end=END_AGE_RANGE;
 label=AGEGROUP;
 if AGEGROUP in (1,2,3) then eexcl="Y";
 else                        eexcl="N";
 if AGEGROUP=4 then hlo="H";
 output;
 if lastcase then do;
  hlo='O';
  label=0;
  output;
 end;
run;

proc format cntlin=fmt;
run;
```

Working with SAS Formats

Using CNTLIN= to Create a Numeric Informat (continued)

Let's use the informat now:

```
data AGEGRPNUM_TEST;
 AGE_KEY=5;AGEGROUP=input(AGE_KEY,A2GN.);output;
 AGE_KEY=14;AGEGROUP=input(AGE_KEY,A2GN.);output;
 AGE_KEY=26;AGEGROUP=input(AGE_KEY,A2GN.);output;
 AGE_KEY=67;AGEGROUP=input(AGE_KEY,A2GN.);output;
 AGE_KEY=-1;AGEGROUP=input(AGE_KEY,A2GN.);output;
run;
```

Obs	AGE_KEY	AGEGROUP
1	5	1
2	14	2
3	26	3
4	67	4
5	-1	0

Note: We'll leave the *Character Informat* scenario to the *Students of Research*.

Working with SAS Formats

Recall Section on Drug Concomitancy (BAND(), BOR(), etc.)

Recall I said in that section in the deck that I would show you how to use SAS Formats to map the drug number ROWNUM to the DRUG_NAME. Here we use a SAS Numeric Format:

```
data UNIQUE_NDCS;
 set UNIQUE_NDCS;
 ROWNUM=_n_;
 DRUG_PWR=2**(ROWNUM-1);
 BIN_REPR=put(DRUG_PWR,binary32.);
 DRUG_NAME="DRUG " || put(ROWNUM,z4.);
run;

data fmt(keep=fmtname hlo type start end label);
 length fmtname $ 8 hlo $ 1 type $ 1 start end 8 label $ 10;
 retain fmtname 'D2N' hlo ' ' type 'N';
 set UNIQUE_NDCS end=lastcase;
 start=ROWNUM;
 end=ROWNUM;
 label=DRUG_NAME;
 output;
 if lastcase then do;
  hlo='O';
  label='** UKNOWN DRUG **';
  output;
 end;
run;
```

Working with SAS Formats

Recall Section on Drug Concomitancy (BAND(), BOR(), etc.)


```
proc format cntlin=fmt;
run;

data FINAL(drop=i);
 length ALL_DRUGS $ 1000;
 attrib DRUG_CONCOM format=binary32.;
 set DRUG_CONCOM;
 ALL_DRUGS="";
 do i=1 to 29;
  if band(DRUG_CONCOM,2**(i-1))>0 then do;
   ALL_DRUGS=catx("-",ALL_DRUGS,put(i,D2N.)); /* Original: "DRUG " || put(i,z3.) */
  end;
 end;
run;
```

SAS Hash and Hash Iterator Objects

SAS Hash and Hash Iterator Objects

Please refer to the following document on the sheepsqueezers.com website:

Introduction to SAS Hash and Hash Iterator Objects

SAS Output Delivery System (ODS)

SAS Output Delivery System (ODS)

Please refer to the following document on the sheepsqueezers.com website:

Introduction to the SAS Output Delivery System (ODS)

Reading and Writing XML Data

Reading and Writing XML Data

Which Hole Did XML Crawl Out Of?

HTML. XML. CSS. DTD. XSD. XSLT. DHTML.

[What, did Elgar write a sequel to the *Enigma Variations*!?! Ha-ha! That's a little musical joke.]

For the past couple of years, we've been bombarded with Internet-related acronyms, like HTML, CSS, etc. One of the newest acronym kids on the block is XML and it allows you to transfer your data, formed in a special "text" format, from one computer platform to another – regardless of operating system or character encoding – without the need for specialized software to read the data, such as SAS for SAS datasets, Excel for Excel workbooks, etc. Before we show you XML and how to read and write it from within a SAS program, we'd like to share with you a brief history of HTML, XML, etc.

In 1991, Tim Berners-Lee created the **H**yper**t**ext **M**arkup **L**anguage (HTML) which is the language used to create web pages. But, HTML is actually created from a much older and more complex markup language called the **S**tandard **G**eneralized **M**arkup **L**anguage (SGML). SGML was invented in the late 1960s at IBM by **G**oldfarb, **M**osher and **L**orris, and was used for years by the publishing industry, the (large-scale) information processing industry, and the government.

Unlike HTML, SGML does NOT specify how text should be presented. That is, there are no `<B>`, `<H1>`, `<TITLE>`, `<ADDRESS>` tags, etc., but SGML is a specification that allows people to *create* their own markup language, such as HTML.

Reading and Writing XML Data

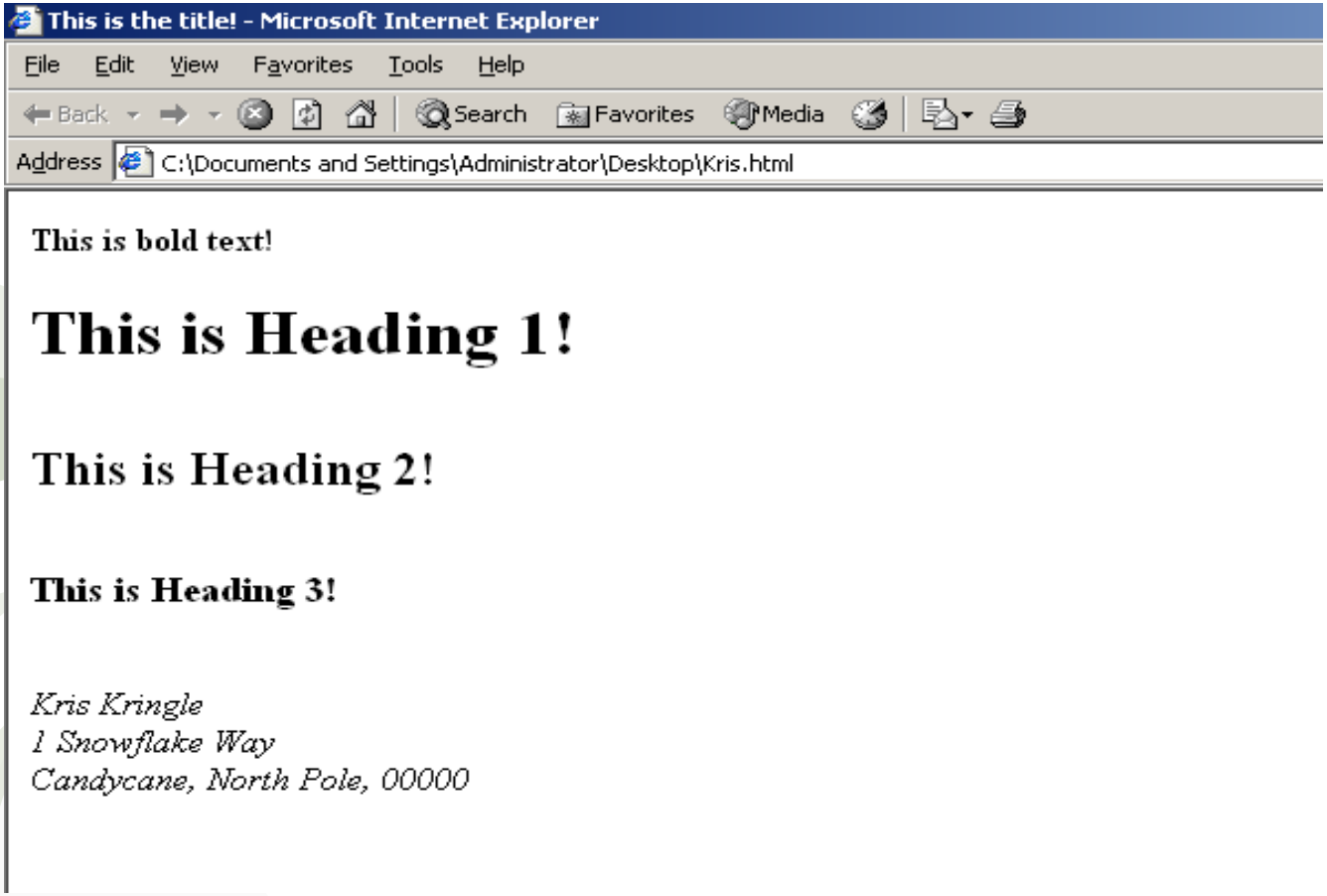
So, is HTML a *formatting language* or a *content language*? Here are those tags again: <B>, <H1>, <TITLE>, <ADDRESS>, etc. Here is an example (see next slide for an example of what a browser would show):

```
<HTML>
 <HEAD>
  <TITLE>This is the title!</TITLE>
 </HEAD>
 <BODY>
  <B>This is bold text!</B> <BR>
  <H1>This is Heading 1!</H1> <BR>
  <H2>This is Heading 2!</H2> <BR>
  <H3>This is Heading 3!</H3> <BR>
  <ADDRESS>
   Kris Kringle <BR>
   1 Snowflake Way <BR>
   Candycane, North Pole, 00000
  </ADDRESS>
 </BODY>
</HTML>
```

Unfortunately, HTML is a bad *formatting language* and a bad *content language*. Why? Assume we were trying to represent NDC codes with their associated USC classes. This information could be represented in a formatted way, but the content – as well as the relationships between NDC and the USCs – gets lost.

SGML's intent was to separate content from formatting, something that has been lost (somewhat) with HTML. Also, with the set of HTML tags available, formatting your web page *as if it were a true printed document* becomes next to impossible.

Reading and Writing XML Data



Reading and Writing XML Data

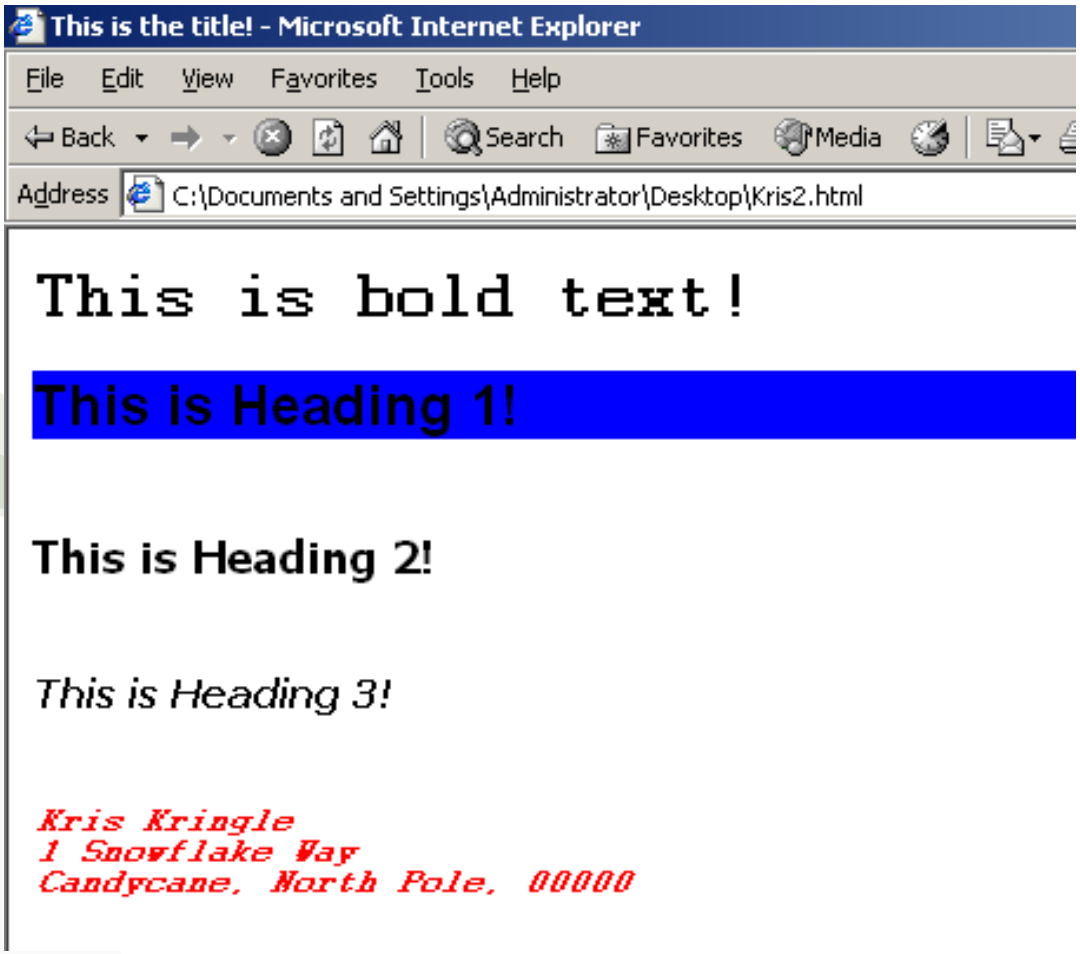
So, in order to allow HTML web pages to be formatted just like a Microsoft Word, Adobe Acrobat PDF, etc. documents, *Cascading Style Sheets* (CSS) was invented as an addition to HTML's lame formatting abilities. Here is that example again (see next slide for an example of what a browser would show):

```
<HTML>
<HEAD>
 <TITLE>This is the title!</TITLE>
</HEAD>
<BODY>
 <B STYLE="font-family:Courier;font-size:20pt">This is bold text!</B> <BR>
 <H1 STYLE="font-family:Arial;font-size:18pt;background:blue">This is Heading 1!</H1> <BR>
 <H2 STYLE="font-family:Tahoma;font-size:14pt">This is Heading 2!</H2> <BR>
 <H3 STYLE="font-family:Verdana;font-style:italic;font-size:12pt">This is Heading 3!</H3> <BR>
 <ADDRESS STYLE="font-family:Courier;font-size:10pt;font-weight:bold;color:red">
  Kris Kringle <BR>
  1 Snowflake Way <BR>
  Candycane, North Pole, 00000
 </ADDRESS>
</BODY>
</HTML>
```

This is all well-and-good, but the problem of *content* still hasn't been addressed, which is where XML comes in. E**x**tensible **M**arkup **L**anguage (XML) emphasizes the importance of content by allowing programmers to create their own set of tags, similar the HTML's `<B>`, `<H1>`, etc., but with **no** ties to textual formatting just content and its relationships.

HTML is an application of SGML whereas XML is a subset of SGML.

Reading and Writing XML Data



Reading and Writing XML Data

Let's show a simple example of XML:

```
<patients>
  <patient patientID="10029">
    <name>Harland, Calvin</name>
    <address>
      <street>18891 SE 90th</street>
      <apt>J253</apt>
      <city>Redmond</city>
      <state>WA</state>
      <zip>98052</zip>
    </address>
    <birthdate>06/30/70</birthdate>
  </patient>
  <patient patientID="10030">
    <name>Grabel, Betty</name>
    <address>
      <street>1222 NE 100th</street>
      <apt></apt>
      <city>Bellevue</city>
      <state>WA</state>
      <zip>98053</zip>
    </address>
    <birthdate>01/09/43</birthdate>
  </patient>
          .
          .
          .
</patients>
```

- HTML is designed to display data and is focused on how data looks, while XML is designed to describe data and to focus on what data is.
- While XML tags can be used to describe the structure of an item such as a purchase order, it does not contain any code that can be used to send that purchase order, process it, or ensure that it is filled. Other people must write code to actually do these things with your XML-formatted data.
- Unlike HTML, *XML tags are defined by the author of a schema or document and are unlimited.* HTML tags are predefined; HTML authors can only use tags that are supported by the current HTML standard.
- Sadly, XML can take up a lot of space!!

Reading and Writing XML Data

Based on the previous example of XML data, where did the tags `PATIENTS`, `PATIENT`, `NAME`, `ADDRESS`, `STREET`, `APT`, `CITY`, `STATE`, `ZIP`, and `BIRTHDATE` come from? The programmer creates a *Document Type Definition* (DTD) or *XML Schema Definition* (XSD), both of which define the tags that are allowed in your XML, as well as the data type, number of occurrences, etc. DTDs are used a lot less over XSDs nowadays. Here is an example of an XSD from the patient data…don't worry, you won't have to create one yourself:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="patients" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="patients" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="patient">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string" minOccurs="0" msdata:Ordinal="0" />
              <xs:element name="birthdate" type="xs:string" minOccurs="0" msdata:Ordinal="2" />
              <xs:element name="address" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="street" type="xs:string" minOccurs="0" />
                    <xs:element name="apt" type="xs:string" minOccurs="0" />
                    <xs:element name="city" type="xs:string" minOccurs="0" />
                    <xs:element name="state" type="xs:string" minOccurs="0" />
                    <xs:element name="zip" type="xs:string" minOccurs="0" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="patientID" type="xs:string" />
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Reading and Writing XML Data

Using SAS to Write XML Data

```
libname SASIN "C:\TEMP\ResultsSASDatasets";
run;

data DrugChain;
 set SASIN.Drug_Chain_Info(obs=10
                           keep=NCPDP_PROVIDER_KEY LocalSite STORE_NUMBER ADDRESS_1
                                City State_Code Zipcode_5D NCPDP_PROVIDER_NUMBER);
run;

libname SASIN clear;
run;

libname XMLOUT xml "C:\TEMP\Drug.xml" xmltype=generic tagset=tagsets.SASXMNSP;
run;

data XMLOUT.DrugChain;
 set DrugChain;
run;

/* This option seems to be faster than the datastep method above. */
proc copy in=WORK out=XMLOUT;
 select DrugChain;
run;

libname XMLOUT clear;
run;
```

Reading and Writing XML Data

Using SAS to Write XML Data


```
<?xml version="1.0" encoding="windows-1252" ?>
<TABLE>
   <DRUGCHAIN>
      <LOCALSITE>STORE #12345</LOCALSITE>
      <NCPDP_PROVIDER_NUMBER>123</NCPDP_PROVIDER_NUMBER>
      <STORE_NUMBER>12</STORE_NUMBER>
      <ADDRESS_1>123 MAIN ST</ADDRESS_1>
      <CITY>ANYCITY</CITY>
      <STATE_CODE>PA</STATE_CODE>
      <ZIPCODE_5D>12345</ZIPCODE_5D>
      <NCPDP_PROVIDER_KEY>1</NCPDP_PROVIDER_KEY>
   </DRUGCHAIN>
 <DRUGCHAIN>
      <LOCALSITE>STORE #12346</LOCALSITE>
      <NCPDP_PROVIDER_NUMBER>124</NCPDP_PROVIDER_NUMBER>
      <STORE_NUMBER>13</STORE_NUMBER>
      <ADDRESS_1>124 MAIN ST</ADDRESS_1>
      <CITY>ANYCITY</CITY>
      <STATE_CODE>PA</STATE_CODE>
      <ZIPCODE_5D>12346</ZIPCODE_5D>
      <NCPDP_PROVIDER_KEY>2</NCPDP_PROVIDER_KEY>
   </DRUGCHAIN>
 <DRUGCHAIN>
      <LOCALSITE>STORE #12347</LOCALSITE>
      <NCPDP_PROVIDER_NUMBER>125</NCPDP_PROVIDER_NUMBER>
      <STORE_NUMBER>14</STORE_NUMBER>
      <ADDRESS_1>125 MAIN ST</ADDRESS_1>
      <CITY>ANYCITY</CITY>
      <STATE_CODE>PA</STATE_CODE>
      <ZIPCODE_5D>12347</ZIPCODE_5D>
      <NCPDP_PROVIDER_KEY>3</NCPDP_PROVIDER_KEY>
   </DRUGCHAIN>
</TABLE>
```

Reading and Writing XML Data

Using SAS to Write XML Data

Let's briefly talk about the XMLTYPE= Libname option. By default, SAS uses the XMLTYPE=GENERIC option when creating an XML file from a SAS dataset. As shown on the previous slide, the XMLTYPE=GENERIC option sets the outer XML tag to the text TABLE and the inner XML tag to the name of the SAS Dataset. The following is a list of XMLTYPE= options:

- GENERIC – outer tag is TABLE, inner tag is the name of the SAS dataset
- ORACLE – outer tag is ROWSET, inner tag is ROW. For use with Oracle XML feature.
- HTML – Creates an HTML TABLE (<TABLE>…</TABLE>).
- MSACCESS – You can export a dataset as XML for use with MS Access 2002/2003.

Let's briefly talk about the TAGSETS= Libname option. By default, when SAS uses the XMLTYPE=GENERIC option, SAS automatically sets TAGSETS=SASXMOG. This tagset sadly leaves a blank space before and after each data element causing your XML file size to grow. You can specify TAGSETS=SASXMNSP to create XML without those additional spaces.

When writing out high-precision numeric data, you may want to add the XML Libname engine option XMLDOUBLE=PRECISION to ensure the best precision.

Reading and Writing XML Data

Using SAS to Write XML Data

Let's create an XML and XSD file for use with Microsoft Access 2002/2003.

```
filename XSDOUT "C:\TEMP\Drug2.xsd";
run;

libname XMLOUT xml "C:\TEMP\Drug2.xml" xmltype=msaccess xmlmeta=schemadata xmlschema=XSDOUT;
run;

proc copy in=WORK out=XMLOUT;
 select DrugChain;
run;

libname XMLOUT clear;
run;
```

Note that both `Drug2.xml` and `Drug2.xsd` are created. The `Drug2.xsd` file is referred to in the `Drug2.xml` file. You can read in the XML file from within Microsoft Access 2002 or 2003. Note that the XSD file contains definitions of the columns, such as data type, etc.

Reading and Writing XML Data

Using SAS to Write XML Data

As you've seen, when the XML file is created, each *column* within a *row* of a SAS dataset becomes a series of XML tags, like this:

```
<DRUGCHAIN>
   <LOCALSITE>STORE #12345</LOCALSITE>
   <NCPDP_PROVIDER_NUMBER>123</NCPDP_PROVIDER_NUMBER>
   <STORE_NUMBER>12</STORE_NUMBER>
   <ADDRESS_1>123 MAIN ST</ADDRESS_1>
   <CITY>ANYCITY</CITY>
   <STATE_CODE>PA</STATE_CODE>
   <ZIPCODE_5D>12345</ZIPCODE_5D>
   <NCPDP_PROVIDER_KEY>1</NCPDP_PROVIDER_KEY>   </DRUGCHAIN>
```

There is an SAS XML Libname option which allows you to write out each column within a row of a SAS dataset as an XML attribute, like this:

```
<DRUGCHAIN>
    <COLUMN name="LocalSite" value="Store #12345" />
    <COLUMN name="NCPDP_Provider_Number" value="123" />
    <COLUMN name="Store_Number" value="12" />
    <COLUMN name="Address_1" value="123 MAIN ST" />
    <COLUMN name="City" value="ANYCITY" />
    <COLUMN name="State_Code" value="PA" />
    <COLUMN name="ZipCode_5D" value="12345" />
    <COLUMN name="NCPDP_PROVIDER_KEY" value="1" />
</DRUGCHAIN>
```

The choice to use depends on the requirement of the target user.

Reading and Writing XML Data

Using SAS to Write XML Data

Here is the Libname statement you can use to create attributes instead of element tags:

```
libname XMLOUT xml "C:\TEMP\Drug_ATTR.xml" xmltype=generic
                                           xmldouble=precision
                                           xmldataform=attribute /* element */
                                           tagset=tagsets.SASXMNSP;
run;

proc copy in=WORK out=XMLOUT;
 select DrugChain;
run;

libname XMLOUT clear;
run;
```

Note: Do not use `XMLTYPE=ORACLE` with `XMLDATAFORM=ATTRIBUTE`…the resulting XML data is incorrect.

Reading and Writing XML Data

Using SAS to Read XML Data

In order to read an XML data file into a SAS dataset via the XML Libname Engine, the XML data has to be in the following format:

```
<?xml version="1.0" encoding="windows-1252" ?>
<TABLE>
          <SAS-dataset-name>
                    <tag1>…</tag1>
                    <tag2>…</tag2>
                    <tag3>…</tag3>
          </SAS-dataset-name>
          <SAS-dataset-name>
                    <tag1>…</tag1>
                    <tag2>…</tag2>
                    <tag3>…</tag3>
          </SAS-dataset-name>
…
</TABLE>
```

Note that each time a *SAS-dataset-name* is encountered, it is treated as a row in the SAS dataset. Note that this is the same format that is created when writing a SAS dataset to an XML file if you use the `XMLTYPE=GENERIC` option.

As an example, let's read in the 10 rows of Drug Chain data we created.

Reading and Writing XML Data

Using SAS to Read XML Data

```
libname XMLIN xml "C:\TEMP\Drug.xml" xmltype=generic xmldouble=precision xmlprocess=relax;
run;

data DrugChainIN;
 set XMLIN.DrugChain;
run;

libname XMLIN clear;
run;
```

Note that character data that looks like numeric data, such as a zip code, may import as a number and you will lose the leading zeroes.

Here we specify the `XMLTYPE=GENERIC` option since that is the format of the XML.

The `XMLPROCESS=RELAX` option allows you to read in XML that has un-escaped double-quotes, single-quotes, and ampersands. Usually, these are replaced by `&quot;`, `&apos;` and `&` upon creation of the XML data.

Reading and Writing XML Data

Using SAS to Read XML Data

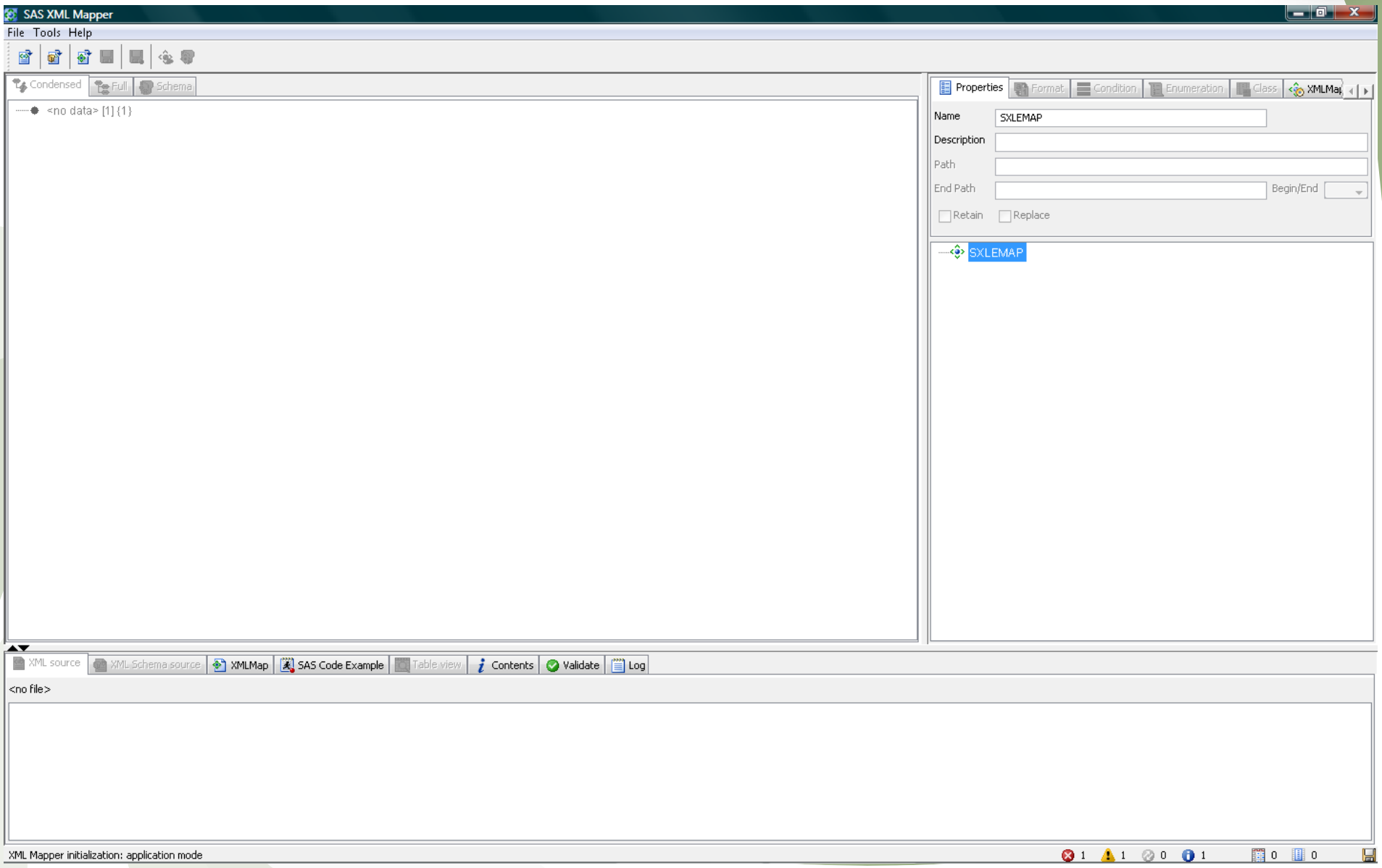
Obviously, not all XML data will be in the generic format described above. Rather than drive everyone crazy by trying to teach you how to use the XMLMap syntax, please refer to the `SAS 9.# XML Libname Engine User's Guide`. XMLMap syntax requires that you know some *XPATH syntax* and is beyond the scope of this discussion.

On a more joyous note, you can use the SAS XML Mapper software to help you create the XMLMap syntax needed to read in non-standard XML. The XML Mapper software is located on the *SAS Client-Side Components Volume 1 CD* or on the SAS Support website.

The next slide shows an example of the SAS XML Mapper software.

Reading and Writing XML Data

sheepsqueezers.com



FTP, URL and EMAIL Options on the FILENAME Statement

FTP, URL and EMAIL Options on the FILENAME Statement

As we've seen in the lecture on reading and writing XML, the LIBNAME statement can take several options that changes the way it behaves. For example, the SAS XML Libname statement is just a regular LIBNAME statement with the XML option entered in. Recall that we were able to export SAS datasets using the XPORT option on the LIBNAME statement. With that said, it should be no surprise that the FILENAME statement has additional useful options as well. We describe three options: FTP, URL and EMAIL. The FTP option allows you to FTP files, the URL option allows you to access web pages, and the EMAIL option allows you to send emails, all from within a SAS program.

You may be more familiar with the FILENAME statement as the statement you use to point to a file on disk that you either want to read from or write to. These actions, of course, still work, but you do not specify FTP, URL, or EMAIL on the FILENAME line.

One note about the FILENAME statement you may not be familiar with. As mentioned above, programmers usually provide the location and name of a file to be read from or written to, like this:

```
filename txtin "C:\TEMP\MyBigFile.txt";
run;

data MyBigData;
 infile txtin truncover;
 input @1 MyData $char100.;
run;

filename txtin clear;
run;
```

FTP, URL and EMAIL Options on the FILENAME Statement

But, you can actually just provide the location of the subdirectory containing your file or files. If you do this, you add the name of your file in parentheses on the INFILE statement:

```
filename txtin "C:\TEMP\"; /* sub-directory now!! */
run;

data MyBigData;
 infile txtin("MyBigFile.txt") truncover;
 input @1 MyData $char100.;
run;

filename txtin clear;
run;
```

FTP, URL and EMAIL Options on the FILENAME Statement

FTP FILENAME Option

In this section, we talk a little about how to FTP files within a SAS program from an FTP site. Many of you may be familiar with the PIPE option on the FILENAME statement which allows access to command line programs, such as FTP.EXE. This method uses the DATA STEP to start the FTP command. Unfortunately, the DATA Step used in this way seems to have a 2-minute time-out. This makes us sad! Luckily, the new SAS FILENAME FTP command does not have this limitation. This makes us glad! Here is an example of how to FTP a file from a remote site and constitute the file locally:

```
filename xfer ftp "remote-filename" host="remote-IP-address" cd="subdirectory" user="username" pass="password"
                  rcmd="binary" recfm=s;
run;

/* RECFM=N indicates binary format and causes the file to be treated as a byte stream */
filename outfile "location-and-filename-of-recreated-file" recfm=n;
run;

data _null_;
 nb=1;
 length abyte $ 1;
 infile xfer nbyte=nb; /* NBYTE=NB means the name of the variable containing the number of bytes to read */
 file outfile recfm=n; /* RECFM=N indicates binary format and causes the file to be treated as a byte stream */
 input abyte $char1.;
 put abyte $char1.;
run;

filename outfile clear;
run;

filename xfer clear;
run;
```

FTP, URL and EMAIL Options on the FILENAME Statement

FTP FILENAME Option

To transfer a file locally to a remote FTP site, the code is nearly the same:

```
filename xfer ftp "remote-filename" host="remote-IP-address" cd="subdirectory" user="username" pass="password"
                  rcmd="binary" recfm=s;
run;

/* RECFM=N indicates binary format and causes the file to be treated as a byte stream */
filename infile "location-and-filename-of-local-file" recfm=n;
run;

data _null_;
 nb=1;
 length abyte $ 1;
 infile infile recfm=n; /* NBYTE=NB means the name of the variable containing the number of bytes to read */
 file xfer nbyte=nb; /* RECFM=N indicates binary format and causes the file to be treated as a byte stream */
 input abyte $char1.;
 put abyte $char1.;
run;

filename infile clear;
run;

filename xfer clear;
run;
```

FTP, URL and EMAIL Options on the FILENAME Statement

URL FILENAME Option

To read in a webpage from a website, you can do this:

```
filename INURL url "http://www.mysite.com/webpage.html";
run;

data WEBSITE;
 infile INURL truncover;
 input @1 aline $char100.;
run;

filename INURL clear;
run;
```

Since you probably won't know the record length ahead of time, you can do something like this:

```
filename INURL url "http://localhost/test1.html";
run;

data WEBSITE;
 infile INURL truncover lrecl=1000 length=len;
 input @1 aline $varying100. len;
run;

filename INURL clear;
run;
```

FTP, URL and EMAIL Options on the FILENAME Statement

EMAIL FILENAME Option

To send an email (with an attachment) from within your SAS program, you can do this:

```
filename mymail email from="DUDLEY@DUDLEYEMAIL.COM"
         to="BOB@BOBMAIL.NET"
         subject="TEST EMAIL"
         content_type="text/html"
         attach="C:\TEMP\Drug.xml";
run;

data _null_;
 file mymail;
 put "ENTER THE ENTIRE BODY OF THE EMAIL IN ONE OR MORE PUT STATEMENTS";
run;

filename mymail clear;
run;
```

Make sure to talk to your SAS System Administrator first before attempting this option. He/she will have to update the SAS Configuration File (e.g., sasv9.cfg) with the appropriate email-related configuration options such as:

-emailsys *fill in your email server system type (such as: MAPI)*
-emailhost *fill in your email server name*
-emailport *fill in your email server port number*

Working With Perl Regular Expressions

Working With Perl Regular Expressions

Perl? What are ya' talkin' about, man?!? This is a presentation on SAS not Perl!!

Perl is a "little" language used all over the world, on diverse types of machines, and on diverse operating systems. It is an interpreted language, handles text wonderfully, and is used for just about everything from administrative tasks to automating web pages.

So what does that have to do with SAS?

One thing Perl has built into it is the ability to used Unix Regular Expressions. You can think of Regular Expressions as grown-up wildcards, like *, % and _. Recall the you use the asterisk to mean "match zero or more characters", the percent sign to mean "match zero or more characters" and the underscore to mean "match exactly one character". For example, ABC*.txt means the letters "ABC" followed by any text (or no text at all) followed by ".txt". The percent sign and underscore are used in SAS, Oracle, and SQL Server SQL along with the `LIKE` operator.

Note that SAS actually has two regular expressions engines built into Base SAS: *SAS Regular Expressions* and *Perl Regular Expressions*. SAS Regular Expressions use syntax that is SAS-specific whereas Perl Regular Expressions uses standard regular expressions syntax found on Unix/Linux machines. This lecture focuses on Perl Regular Expressions and NOT SAS Regular Expressions.

By the way, you can easily distinguish functions that are Perl Regular Expressions from SAS Regular Expressions. Perl Regular Expression functions begin with PRX, where as SAS Regular Expression functions begin with RX.

Working With Perl Regular Expressions

Before we talk about Regular Expressions, let's remind ourselves about the LIKE operator in SQL. Recall that you can use the underscore (_) to match exactly one character and the percent (%) to match zero or more characters. For example,

```
SELECT *
 FROM MYDRUGBRANDS
 WHERE DRUG_NAME LIKE '_EXX%';
```

would find all rows for the drug LEXXEL.

And the following would find all drugs beginning the LE and ending with EL:

```
SELECT *
 FROM MYDRUGBRANDS
 WHERE DRUG_NAME LIKE 'LE__EL';
```

Drugs like: LEXXEL, LEMMEL, LEVVEL, LETTEL, etc. LEEL is NOT matched.

The underscore and the percent are nice, but are limited…

Working With Perl Regular Expressions

Regular Expression Operators and Metasymbols Operator Description


```
\ marks the next character as either a special character, a literal, a back reference, or an octal escape:
        "n" matches the character "n"
        "\n" matches a new line character
        "\\" matches "\"
        "\(" matches "("

| specifies the or condition when you compare alphanumeric strings.

^ matches the position at the beginning of the input string.

$ matches the position at the end of the input string.

* matches the preceding subexpression zero or more times:
        zo* matches "z" and "zoo"
        * is equivalent to {0}

+ matches the preceding subexpression one or more times:
        "zo+" matches "zo" and "zoo"
        "zo+" does not match "z"
        + is equivalent to {1,}

? matches the preceding subexpression zero or one time:
        "do(es)?" matches the "do" in "do" or "does"
        ? is equivalent to {0,1}

{n} n is a non-negative integer that matches exactly n times:
        "o{2}" matches the two o's in "food"
        "o{2}" does not match the "o" in "Bob"

{n,} n is a non-negative integer that matches n or more times:
        "o{2,}" matches all the o's in "fooooood"
        "o{2,}" does not match the "o" in "Bob"
        "o{1,}" is equivalent to "o+"
        "o{0,}" is equivalent to "o*"

{n,m} m and n are non-negative integers, where n<=m. They match at least n and at most m times:
        "o{1,3}" matches the first three o's in "fooooood"
        "o{0,1}" is equivalent to "o?"
Note: You cannot put a space between the comma and the numbers.
```

Working With Perl Regular Expressions

period (.) matches any single character except newline. To match any character including newline, use a pattern such as "[.\n]".

(pattern) matches a pattern and captures the match. To retrieve the position and length of the match that is captured, use CALL PRXPOSN. To match parentheses characters, use "\(" or "\)".

x|y matches either x or y:
"z|food" matches "z" or "food"
"(z|f)ood" matches "zood" or "food"

[xyz] specifies a character set that matches any one of the enclosed characters:
"[abc]" matches the "a" in "plain"

[^xyz] specifies a negative character set that matches any character that is not enclosed:
"[^abc]" matches the "p" in "plain"

[a-z] specifies a range of characters that matches any character in the range:
"[a-z]" matches any lowercase alphabetic character in the range "a" through "z"

[^a-z] specifies a range of characters that does not match any character in the range:
"[^a-z]" matches any character that is not in the range "a" through "z"

\b matches a word boundary (the position between a word and a space):
"er\b" matches the "er" in "never"
"er\b" does not match the "er" in "verb"

\B matches a non-word boundary:
"er\B" matches the "er" in "verb"
"er\B" does not match the "er" in "never"

\d matches a digit character that is equivalent to [0-9].

\D matches a non-digit character that is equivalent to [^0-9].

\s matches any white space character including space, tab, form feed, and so on, and is equivalent to [\f\n\r\t\v].

\S matches any character that is not a white space character and is equivalent to [^\f\n\r\t\v].

\t matches a tab character and is equivalent to "\x09".

Working With Perl Regular Expressions


```
\w matches any word character including the underscore and is equivalent to [A-Za-z0-9_].

\W matches any non-word character and is equivalent to [^A-Za-z0-9_].

\num matches num, where num is a positive integer. This is a reference back to captured matches:
        "(.)\1" matches two consecutive identical characters.
```

There are several functions and call routines associated with Perl Regular Expressions:

- PRXMATCH() Function
- PRXPAREN() Function
- PRXPARSE() Function
- CALL PRXCHANGE() Routine
- CALL PRXNEXT() Routine
- CALL PRXPOSN() Routine
- CALL PRXSUBSTR() Routine

We will explain each function and call routine during the course of this lecture.

Working With Perl Regular Expressions

First, let's try to become comfortable with the regular expressions syntax.

Examples of Regular Expression

1. Match one or more characters:
   ```
   .*
   ```
2. Match the letter A followed by zero, one, or more characters:
   ```
   A.*
   ```
3. Match the letter A followed by a number:
   ```
   A[0123456789]
   ```
4. Match the letter A followed by a single number followed by more stuff:
   ```
   A[0123456789].*
   ```
5. Match the letter A followed by exactly two numbers:
   ```
   A[0123456789]{2} or A\d{2}
   ```
6. Match the letter A followed by at least two numbers but not more than 4 number:
   ```
   A\d{2,4}
   ```
7. Match one or more letters followed by two or more numbers:
   ```
   [a-zA-Z]{1,}\d{2,}
   ```
8. Match a social security number:
   ```
   ^\d{3}-\d{2}-\d{4}$
   ```
9. Match LEXXEL 50MG BOTTLE 100:
   ```
   ^LEXXEL \d{1,}MG BOTTLE \d{1,}$
   ```
10. Same as #9, but use grouping expressions to capture the value before the *MG*:
    ```
    ^LEXXEL (\d{1,})MG BOTTLE \d+$
    ```
    referred to as **\1**

Working With Perl Regular Expressions

Here's an example validating social security numbers using Perl Regular Expressions:

```
data GoodSSN(drop=reSSN)
     BadSSN(drop=reSSN);
 retain reSSN;
 infile cards truncover;
 input @1 SSN $char11.;

 if _N_=1 then do;

  /* Parse the Social Security Number Regular Expression */
  reSSN=PRXPARSE("/\d{3}-\d{2}-\d{4}/");

  /* Check that the regular expression parsed properly */
  if missing(reSSN) then do;
   putlog "ERROR: reSSN failed to parse properly!";
   stop;
  end;

 end;

 /* At this point, we can validate each Social Security Number */
 if PRXMATCH(reSSN,SSN) then output GoodSSN;
 else                        output BadSSN;

cards;
123-45-6789
234-56-7890
1234-564-09
345-67-8901
;
run;
```

PRXMATCH(re,variable) returns zero if there was no match; otherwise it returns the starting position if a match was found.

Working With Perl Regular Expressions

Here's another example validating the label name for LEXXEL:

```
data GoodLN(drop=reLN)
     BadLN(drop=reLN);
 retain reLN;
 infile cards truncover;
 input @1 LN $char30.;

 if _N_=1 then do;

  /* Parse the Label Name Regular Expression */
  reLN=PRXPARSE("/LEXXEL \d{1,}MG BOTTLE \d{1,}/");

  /* Check that the regular expression parsed properly */
  if missing(reLN) then do;
   putlog "ERROR: reLN failed to parse properly!";
   stop;
  end;

 end;

 /* At this point, we can validate each label name */
 if PRXMATCH(reLN,LN) then output GoodLN;
 else                      output BadLN;

cards;
LEXXEL 50MG BOTTLE 100
LEXXEL 25MG BOTTLE 200
LEXXEL 15MG BOTTLE 300
LEXXEL MG BOTTLE
;
run;
```

PRXMATCH returns zero if no match was found; otherwise, the position of the match.

Working With Perl Regular Expressions

Here's an example that pulls the MG value:


```
data GoodLN(drop=reLN)
     BadLN(drop=reLN);
 retain reLN;
 infile cards truncover;
 input @1 LN $char30.;

 if _N_=1 then do;

  /* Parse the Label Name Regular Expression */
  reLN=PRXPARSE("/LEXXEL (\d{1,})MG BOTTLE \d{1,}/");

  /* Check that the regular expression parsed properly */
  if missing(reLN) then do;
   putlog "ERROR: reLN failed to parse properly!";
   stop;
  end;

 end;

 /* At this point, we can validate each label name */
 if PRXMATCH(reLN,LN) then do;

  /* Determine the starting position and length of the strength */
  CALL PRXPOSN(reLN,1,pos,len);

  /* Pull the MG number */
  Milligrams=input(substr(LN,pos,len),2.);

  output GoodLN;
 end;
 else output BadLN;

cards;
LEXXEL 50MG BOTTLE 100
LEXXEL 25MG BOTTLE 200
LEXXEL 15MG BOTTLE 300
LEXXEL MG BOTTLE
LEXXEL 5MG BOTTLE 300
;
run;
```

`CALL PRXPOSN(re,#,start-pos,len)` returns the starting position and length of the #th grouping pattern.

Obs	LN	pos	len	Milligrams
1	LEXXEL 50MG BOTTLE 100	20	3	50
2	LEXXEL 25MG BOTTLE 200	20	3	25
3	LEXXEL 15MG BOTTLE 300	20	3	15
4	LEXXEL 5MG BOTTLE 300	19	3	5

Working With Perl Regular Expressions

Here's an example that pulls the package count value (code continued on next slide):

```
data GoodLN(drop=reLN)
     BadLN(drop=reLN);
 retain reLN;
 infile cards truncover;
 input @1 LN $char30.;

 if _N_=1 then do;

  /* Parse the Label Name Regular Expression */
  reLN=PRXPARSE("/LEXXEL (\d{1,})MG BOTTLE (\d{1,})/");

  /* Check that the regular expression parsed properly */
  if missing(reLN) then do;
   putlog "ERROR: reLN failed to parse properly!";
   stop;
  end;

 end;

 /* At this point, we can validate each label name */
 if PRXMATCH(reLN,LN) then do;

  /* Determine the starting position and length of the strength */
  CALL PRXPOSN(reLN,1,pos,len);

  /* Pull the MG number */
  Milligrams=input(substr(LN,pos,len),2.);

  /* Determine the starting position and length of the package count */
  CALL PRXPOSN(reLN,2,pos,len);

  /* Pull the package count */
  PackageCount=input(substr(LN,pos,len),3.);

  output GoodLN;
 end;
 else output BadLN;
```

Working With Perl Regular Expressions

(continued from previous slide)


```
cards;
LEXXEL 50MG BOTTLE 100
LEXXEL 25MG BOTTLE 200
LEXXEL 15MG BOTTLE 300
LEXXEL MG BOTTLE
LEXXEL 5MG BOTTLE 300
;
run;
```

Obs	LN	pos	len	Milligrams	PackageCount
1	LEXXEL 50MG BOTTLE 100	20	3	50	100
2	LEXXEL 25MG BOTTLE 200	20	3	25	200
3	LEXXEL 15MG BOTTLE 300	20	3	15	300
4	LEXXEL 5MG BOTTLE 300	19	3	5	300

Working With Perl Regular Expressions

Here's an example that uses PRXPAREN():

```
data BrandRanking(drop=reBrandRank);
 retain reBrandRank;
 infile cards truncover;
 input @1 BrandName $char10.;

 if _N_=1 then do;

  /* Parse the Brand Rank Regular Expression */
  reBrandRank=PRXPARSE("/(CIALIS)|(ASPIRIN)|(SAMPLEX)|(LEXXEL)/");

  /* Check that the regular expression parsed properly */
  if missing(reBrandRank) then do;
   putlog "ERROR: reBrandRank failed to parse properly!";
   stop;
  end;

 end;

 BrandRank=0;

 if PRXMATCH(reBrandRank,BrandName) then do;

  /* Determine the Brand Ranking */
  BrandRank=PRXPAREN(reBrandRank);

 end;

cards;
LEXXEL
SAMPLEX
ASPIRIN
CIALIS
DUNGBALL
;
run;
```

`PRXPAREN(re)` returns the # of the grouping pattern if there was a match.

Obs	BrandName	BrandRank
1	LEXXEL	4
2	SAMPLEX	3
3	ASPIRIN	2
4	CIALIS	1
5	DUNGBALL	0

Working With Perl Regular Expressions

Here's an example that uses CALL PRXCHANGE():

```
data CatInTheHat(drop=reAT);
 length Sentence NewSentence $ 100;
 retain reAT;

 if _N_=1 then do;

  /* Parse the Brand Rank Regular Expression */
  reAT=PRXPARSE("s/at/og/");

  /* Check that the regular expression parsed properly */
  if missing(reAT) then do;
   putlog "ERROR: reAT failed to parse properly!";
   stop;
  end;

 end;

 Sentence="The big fat cat wearing a hat sat on a mat swatting bats!";
 CALL PRXCHANGE(reAT,-1,Sentence,NewSentence);

 output;

run;
```

Results:

```
The big fog cog wearing a hog sog on a mog swogting bogs!
```

`CALL PRXCHANGE(re,-1,text)` changes `text` based on the substitution regular expression in `re`. The `-1` indicates that ALL changes are to be made throughout `text`.

Working With Perl Regular Expressions

Here's an example that uses CALL PRXSUBSTR():

```
data BrandStrength(drop=reSTR);
 length Strength $ 100;
 retain reSTR;
 infile cards truncover;
 input @1 LN $char30.;

 if _N_=1 then do;

  /* Parse the Strength Regular Expression */
  reSTR=PRXPARSE("/\d{0,}MG/");

  /* Check that the regular expression parsed properly */
  if missing(reSTR) then do;
   putlog "ERROR: reSTR failed to parse properly!";
   stop;
  end;

 end;

 Strength="No Strength";

 /* At this point, we can validate each label name */
 if PRXMATCH(reSTR,LN) then do;

  /* Determine the starting position and length of the strength */
  CALL PRXSUBSTR(reSTR,LN,pos,len);

  /* Pull the MG number */
  Strength=substr(LN,pos,len);

 end;

cards;
LEXXEL 50MG BOTTLE 100
LEXXEL 25MG BOTTLE 200
LEXXEL 15MG BOTTLE 300
LEXXEL MG BOTTLE
LEXXEL 5MG BOTTLE 300
;
run;
```

`CALL PRXSUBSTR(re,text,pos,len)` returns the position `pos` and length `len` of the matching RegEx `re` within `text`.

Obs	Strength	LN	pos	len
1	50MG	LEXXEL 50MG BOTTLE 100	8	4
2	25MG	LEXXEL 25MG BOTTLE 200	8	4
3	15MG	LEXXEL 15MG BOTTLE 300	8	4
4	MG	LEXXEL MG BOTTLE	8	2
5	5MG	LEXXEL 5MG BOTTLE 300	8	3

Working With Perl Regular Expressions

Here's an example that uses CALL PRXNEXT():

```
data PatientDrugs(drop=reDRUG);
 length Drug $ 100;
 retain reDRUG;
 infile cards truncover;
 input @1 PATIENT_KEY          3.
       @5 DRUGLIST     $char100.;

 if _N_=1 then do;

  /* Parse the drug Regular Expression */
  reDRUG=PRXPARSE("/\w*;/");

  /* Check that the regular expression parsed properly */
  if missing(reDRUG) then do;
   putlog "ERROR: reDRUG failed to parse properly!";
   stop;
  end;

 end;

 Drug="**NONE**";
 start=1;

 /* At this point, do we have a match? */
 CALL PRXNEXT(reDRUG,start,-1,DRUGLIST,pos,len);
 do while(pos>0);
  Drug=substr(DRUGLIST,pos,len-1);
  CALL PRXNEXT(reDRUG,start,-1,DRUGLIST,pos,len);
  output;
 end;

cards4;
001 SAMPLEX;NEXIUM;CIALIS;ASPIRIN;RIBOFLAVIN;
;;;;
run;
```

`CALL PRXNEXT(re,start,-1,text,pos,len)` returns the position `pos` and length `len` of the matching RegEx `re` within `text`, starting at `start`. Continued calls to the PRXNEXT routine gets you the next match based on `re` within `text`.

Working With Perl Regular Expressions

Obs	Drug	PATIENT_KEY	DRUGLIST	start	pos	len
1	SAMPLEX	1	SAMPLEX;NEXIUM;CIALIS;ASPIRIN;RIBOFLAVIN;	17	10	7
2	NEXIUM	1	SAMPLEX;NEXIUM;CIALIS;ASPIRIN;RIBOFLAVIN;	24	17	7
3	CIALIS	1	SAMPLEX;NEXIUM;CIALIS;ASPIRIN;RIBOFLAVIN;	32	24	8
4	ASPIRIN	1	SAMPLEX;NEXIUM;CIALIS;ASPIRIN;RIBOFLAVIN;	43	32	11
5	RIBOFLAVIN	1	SAMPLEX;NEXIUM;CIALIS;ASPIRIN;RIBOFLAVIN;	43	0	0

Using Indexes in SAS

Using Indexes In SAS

Occasionally, you will deal with SAS Datasets that are very large, so much so that the act of subsetting the dataset could take quite a while. And heaven forbid if you have to subset over and over again, say, by NDC_KEY, DX_CODE or PRC_CODE, etc. This is where indexing your SAS dataset *may* come in handy. We say "may" because indexing a SAS dataset, while it can be used to avoid the PROC SORTs when going to MERGE two datasets, works best when you want to find a small subset of your dataset based on one or more variables. "Small" here refers to something like less than 15% to 20% of the rows of the dataset will be returned. Any more than that and the index won't help much.

For large datasets that need to have information added to them, you should look into the SAS Hash Object rather than trying to sort the large dataset.

So, what is an Index? An index stores values for a specific variable or variables as well as information as to where that particular information is located within a SAS Dataset. SAS creates a separate index file (.sas7bndx) for the corresponding SAS Dataset (.sas7bdat) and uses that file to determine the location of your data in the SAS dataset itself. This avoids having to sequentially process the SAS Dataset.

There are three ways to create an index in SAS:

- Using the INDEX= SAS Dataset Option on the outgoing SAS Dataset
- Using PROC DATASETS to create the index
- Using PROC SQL (in a similar way we create indexes in Oracle or SQL Server).

Using Indexes In SAS

Before we move on to explaining these three methods, let's talk about the types of indexes we can create:

1. Simple Index – this is an index on only one variable
2. Composite Index – this is an index on two or more variables

For example, we can create a simple index on NDC_KEY if that variable is the only variable with which we subset the dataset. Think WHERE Clause here!

On the other hand, if we need to subset the dataset by, say, NDC_KEY and PATIENT_GENDER, simultaneously, then you need to create a composite index.

A *simple index* is named by using the same name as the variable you want to index, say, NDC_KEY.

A *composite index* is named by giving an index name, say ixNDCSEX, to the variables you want to index.

Simple index and composite index variables are called *key variables*.

Note that the index is *never* used with a Subsetting-IF Statement, only WHERE Clauses.

Don't be tempted to create an index on every variable or combination of variables! The index file will become huge…possibly bigger than the dataset itself!

When using indexes, it's probably a good idea to turn the SAS Message Level to I:

```
options msglevel=I;
```

You will receive informative information on what index was used.

Using Indexes In SAS

Using the INDEX= SAS Dataset Option on the outgoing SAS Dataset

You can create a simple or composite index on the outgoing SAS dataset, that is, a dataset specified on the DATA statement, by using the `INDEX=` dataset option.

For example, if I want to create a *simple index* on the variable NDC_KEY in my new dataset, I can do this (note that I am using the variable name):

```
data MyNewDataset(INDEX=(NDC_KEY));
 set MyOldDataset;
run;
```

If I want to create a *composite index* on NDC_KEY and BIN_KEY, I can do this:

```
data MyNewDataset(INDEX=(idxNDCBIN=(NDC_KEY BIN_KEY));
 set MyOldDataset;
run;
```

When I want to use the index, I can do this:

```
data MyOneNDCCode;
 set MyNewDataset(where=(NDC_KEY='11111111111'));
run;
```

Using Indexes In SAS

Using PROC DATASETS to Create the Index

You can create a simple or composite index on any SAS dataset by using PROC DATASETS.

For example, if I want to create a *simple index* on the variable NDC_KEY in my new dataset, I can do this (note that I am using the variable name):

```
proc datasets library=work;
 modify MyNewDataset;
  index create NDC_KEY;
run;
quit;
```

If I want to create a *composite index* on NDC_KEY and BIN_KEY, I can do this:

```
proc datasets library=work;
 modify MyNewDataset;
 index create idxNDCBIN=(NDC_KEY BIN_KEY);
run;
quit;
```

Note that you can delete an index as well using PROC DATASETS:

```
proc datasets library=work;
 modify MyNewDataset;
 index delete idxNDCBIN;
run;
quit;
```

Using Indexes In SAS

Using PROC SQL to Create the Index

You can create a simple or composite index on any SAS dataset by using PROC SQL.

For example, if I want to create a *simple index* on the variable NDC_KEY in my new dataset, I can do this (note that I am using the variable name):

```
proc sql noprint;
 create index NDC_KEY on MyNewDataset(NDC_KEY);
 drop index IDX1;
quit;
```

If I want to create a *composite index* on NDC_KEY and BIN_KEY, I can do this:

```
proc sql noprint;
 create index idxNDCBIN on MyNewDataset(NDC_KEY,BIN_KEY);
 drop index IDX1;
quit;
```

Note that you can delete an index as well using PROC DATASETS:

```
proc sql noprint;
 drop index idxNDCBIN from MyNewDataset;
run;
quit;
```

Using Indexes In SAS

Example

This example uses a SAS Dataset with nearly 48 million rows. The SAS Dataset file (.sas7bdat) itself is 3GB in size. We show the difference in time to create a new dataset when sequentially searching through a SAS dataset for specific values of two variables versus creating an index on those two variables (composite) and then searching. Note that we will be concentrating on two variables, AT and AG:

AT	Frequency	Percent	Cumulative Frequency	Cumulative Percent
DRUGTYPE_1	38313486	80.22	38313486	80.22
DRUGTYPE_2	9444135	19.78	47757621	100.00

AG	Frequency	Percent	Cumulative Frequency	Cumulative Percent
DRUG_1	5803691	12.15	5803691	12.15
DRUG_2	363383	0.76	6167074	12.91
DRUG_3	2701567	5.66	8868641	18.57
DRUG_4	1154753	2.42	10023394	20.99
DRUG_5	4241121	8.88	14264515	29.87
DRUG_6	5783713	12.11	20048228	41.98
DRUG_7	768680	1.61	20816908	43.59
DRUG_8	5598134	11.72	26415042	55.31
DRUG_9	5266334	11.03	31681376	66.34
DRUG_10	5483042	11.48	37164418	77.82
DRUG_11	1517336	3.18	38681754	81.00
DRUG_12	2638889	5.53	41320643	86.52
DRUG_13	6436978	13.48	47757621	100.00

Using Indexes In SAS

Example

Here is the code used to create the two simple indexes:

```
proc sql noprint;
 create index AT on MYDATA(AT);
 create index AG on MYDATA(AG);
quit;
```

Note that this took about 5 minutes to run. Now, as you saw in the previous slide, DRUG_1 accounts for 0.76% of the rows of data which is less than the 15% limit, so let's pull this data:

```
data DRUG_2;
 set MYDATA(where=(AG='DRUG_2'));
run;
```

Note that if you turned MSGLEVEL to I, you will see this note in the SAS Log:

```
INFO: Index AG selected for WHERE clause optimization.
```

This DATA STEP took 2.68 seconds to run.

Let's do it again using a subsetting IF which does NOT use indexes:

```
data DRUG_2;
 set MYDATA;
 if AG='DRUG_2'; /* No Indexes are used with subsetting-IF Statements! */
run;
```

This Data Step took a little more than 24 seconds to run!

Using Indexes In SAS

Example

Next, let's look at what happens when you choose to subset the data on a value associated with more than 15% of the data, say AT=ANTIDEPRESSANT:

```
/* Ran in 58.59 seconds…no index used */
data DRUGDATA_1;
 set MYDATA(where=(AT=DRUGTYPE_1'));
run;

/* Ran in 47.90 seconds…no index used due to subsetting-IF */
data DRUGDATA_2;
 set MYDATA;
 if AT='DRUGTYPE_1';
run;

/* Ran in 57.75 seconds…index forced not to be used */
data DRUGDATA_3;
 set MYDATA(idxwhere=no where=(AT='DRUGTYPE_1'));
run;
```

Note how DRUGDATA_1 and DRUDATA_3 took about the same time to run…DRUGDATA_1 is NOT using an index and there is no message in the SAS Log indicating that it did. So, if you are pulling back a large percentage of the data, sequential processing is best and it seems that, at least for this example, the subsetting IF statement is fastest.

Note that you can use the SAS Dataset Option `IDXWHERE=YES|NO` to tell SAS to use an index (YES) or not (NO). Also, you can force SAS to use a specific index (against its better judgment) by using `IDXNAME=index-name`.

Locking and Unlocking SAS Datasets

Locking and Unlocking SAS Datasets

Most of the time you create and delete your own SAS datasets. No one else uses them except for you and you never have to worry about *contention*. But, there are times when you need to create a SAS Dataset that will be shared with a group of programmers who may access your dataset at any time. This is hazardous especially if you want to update your dataset in a SAS program that may take a while. That is, you have a SAS Dataset that you want to update. The program to update it is very large containing several Data and Procedure steps. If you do not use a LOCK on the dataset, and someone else swoops in and begins to read the file, you're screwed!

To lock a SAS Dataset, you use the LOCK Statement:

```
LOCK LIBREF.MEMBER-NAME;
```

To unlock a SAS Dataset, use the CLEAR option:

```
LOCK LIBREF.MEMBER-NAME CLEAR;
```

To determine if you have a lock on a SAS Dataset, use the QUERY option:

```
LOCK LIBREF.MEMBER-NAME QUERY;
```

For example, say you wanted to update a SAS Dataset with more timely information. First you need to obtain a lock on the dataset to ensure no one else accesses it; second you need to update the dataset; finally, you need to unlock the dataset.

Locking and Unlocking SAS Datasets

Example:

```
options msglevel=I;

data VAULT.MyImportantDataset;
 A=1;output;
run;

LOCK VAULT.MyImportantDataset;

proc sql noprint;
 update VAULT.MyImportantDataset
  set A=2;
quit;

LOCK VAULT.MyImportantDataset CLEAR;
```

The log file would look like this:

```
15          data VAULT.MyImportantDataset;
16           A=1;output;
17          run;

NOTE: The data set VAULT.MYIMPORTANTDATASET has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


18
19          LOCK VAULT.MyImportantDataset;
NOTE: VAULT.MYIMPORTANTDATASET.DATA is now locked for exclusive access by you.
20
```

Locking and Unlocking SAS Datasets


```
21          proc sql noprint;
22           update VAULT.MyImportantDataset
23            set A=2;
NOTE: 1 row was updated in VAULT.MYIMPORTANTDATASET.

24          quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds


25
26          LOCK VAULT.MyImportantDataset CLEAR;
NOTE: VAULT.MYIMPORTANTDATASET.DATA is no longer locked by you.
```

SAS has cleverly created an Automatic Macro Variable SYSLCKRC which is the return code from a LOCK statement. You can use this return code to determine if you really were able to obtain a lock. If SYSLCKRC is zero, then the lock worked. For example,

```
%if &SYSLCKRC. eq 0 %then %do;

 proc sql noprint;
  update VAULT.MyImportantDataset
   set A=2;
 quit;

 LOCK VAULT.MyImportantDataset CLEAR;

%end;
%else %do;

 %put >>>> FILE CANNOT BE LOCKED!!  NO UPDATE PERFORMED!! <<<<;

%end;
```

Locking and Unlocking SAS Datasets

Note that if you cannot obtain a lock, the SAS Log will reflect this fact:

```
ERROR: You cannot reopen VAULT.MYIMPORTANTDATASET.DATA for update access with member-level
control because VAULT.MYIMPORTANTDATASET.DATA is locked for exclusive access by you in resource
environment IOM ROOT COMP ENV.
```

Generation Data Sets

(It's All History, man!)

Generation Data Sets

There are times when you need to keep historical data in a SAS dataset, but you don't necessarily want to create a "historical date" variable indicating which month (or week, or year, etc.), each row comes from. Instead, you may prefer to create historical datasets, one representing the most recent month, another representing last month, another representing the month before that, etc. This has the advantage of keeping the same format saving you from creating a historical date variable and saving you from updating your code to take into account this new variable.

SAS can do this via *Generation Data Sets*. You can specify the total number of generations (called the *generation number*) you want to keep, and each time you update the original dataset, each generation data set moves down one in line. The set of generation data sets is called a *generation group*.

For example, say we have a SAS Dataset containing a list of script counts per month for each NDC_KEY. The dataset is called NDCRxCounts and contains two variables: NDC_KEY and RX_COUNTS. Now, the first time we run this we create NDCRxCounts. Next month, we run it again, but since NDCRxCounts is a generation data set, SAS renames the current month to NDCRXCOUNTS#A001, and the new dataset is called NDCRXCOUNTS.

The second time do this, NDCRXCOUNTS#A001 is renamed NDCRXCOUNTS#A002, the dataset NDCRXCOUNTS is renamed to NDCRXCOUNTS#A001 and the current month's data is called NDCRXCOUNTS.

Note that SAS dataset names are limited to 28 characters instead of 32!!!

Generation Data Sets

The first time you create a dataset you want to be treated as a generation data set, you must specify the dataset option GENMAX=# on the outgoing dataset, where # is the maximum number of generation data sets you want in your generation group. For example, if you want 24 months of history, set GENMAX=24. In this case, you will have one most-current dataset NDCRXCOUNTS (called the *base version*), and 24 historical datasets called NDCRXCOUNTS#A023 to NDCRXCOUNTS#A001. The maximum number you can specify for GENMAX is 1000.

For example,

```
data VAULT.NDCRXCOUNTS(genmax=24);
 set New_NDCRXCOUNTS;
run;
```

Note that when you specify GENMAX=24 it means *keep one current version and 23 historical versions for a total of 24 versions*: NDCRXCOUNTS, NDCRXCOUNTS#A023 to NDCRXCOUNTS#A001.

Also, note that SAS does NOT peg the names to NDCRXCOUNTS#023 to NDCRXCOUNTS#A001 if you run your program more the 23 times, but will continue to increment the names. For example, NDCRXCOUNTS#024 to NDCRXCOUNTS#A002, then NDCRXCOUNTS#A025 to NDCRXCOUNTS#A003, etc.

This may seem like a pain, but you are **not** supposed to refer to these historical datasets by name, but by *generation number*. You use the dataset option GENNUM=#, where # is a negative number. GENNUM=-1 refers to the most-recent historical generation dataset. GENNUM=0 refers to the current (base) version.

Generation Data Sets

Example (creating generation data set the first time):

```
/* Get this month`s summarized data */
proc sql noprint;
 CREATE TABLE NEW_NDCRXCOUNTS AS
  SELECT NDC_KEY,COUNT(*) AS RX_COUNTS
   FROM MYDRUGDATA
   GROUP BY NDC_KEY;
run;

/* Initial creation of generation data set */
data SASOUT.NDCRXCOUNTS(genmax=24);
 set NEW_NDCRXCOUNTS;
run;

/* Next and subsequent months, get new month`s summarized data */
proc sql noprint;
 CREATE TABLE NEW_NDCRXCOUNTS AS
  SELECT NDC_KEY,COUNT(*) AS RX_COUNTS
   FROM MYDRUGDATA
   GROUP BY NDC_KEY;
run;

/* Over-write dataset to create next generation */
data SASOUT.NDCRXCOUNTS;
 set NEW_NDCRXCOUNTS;
run;
```

Generation Data Sets

Example (using the generation data sets to get the newest NDC codes):

```
proc sql noprint;
 CREATE TABLE NEW_NDCS AS
  SELECT NDC_KEY
   FROM SASOUT.NDCRXCOUNTS
  EXCEPT
  SELECT DISTINCT NDC_KEY
   FROM (SELECT NDC_KEY
          FROM SASOUT.NDCRXCOUNTS(GENNUM=-1)
         UNION
         SELECT NDC_KEY
          FROM SASOUT.NDCRXCOUNTS(GENNUM=-2)
         UNION
         SELECT NDC_KEY
          FROM SASOUT.NDCRXCOUNTS(GENNUM=-3)
        );
quit;
```

There's a lot more on this. Please see the SAS Concepts manual.

Reading from and Writing to Microsoft Excel and Access

Reading from and Writing to Microsoft Excel and Access

Most of you already know how to create Microsoft Excel spreadsheets from within SAS, but since there are two different methods you can use depending on whether you're using a 32-bit SAS server or a 64-bit SAS server, it seems like a good idea to put the differences down on paper.

First, why are there differences between a 32-bit and 64-bit SAS server, you ask? I can hear you say that this is not very "SAS-like"; that is, with SAS you can usually run code on one machine and then very easily run it on another machine without much or any code changes. The problem occurs because the driver – the software responsible for communicating with Microsoft Excel and Microsoft Access – *Jet OLEDB Provider* is not available for 64-bit machines, but is available for 32-bit machines. The decision not to compile the *Jet OLEDB Provider* on 64-bit platforms was Microsoft's and SAS can't do a damn thing about it. So, because of this, we can use *SAS/Access to OLEDB* to read from and write to Excel workbooks and Access databases only on a 32-bit SAS server, but we have to use *SAS/Access to PC Files* on a 64-bit SAS server.

You also may be asking: What about `PROC IMPORT` and `PROC EXPORT`? Good Question! I have not had much luck with these procedures and avoid them if at all possible. Note that if you are running SAS on your local PC and have *SAS/Access to PC Files* installed, `PROC IMPORT` and `PROC EXPORT` work fine and you can use them.

Another reason to avoid `PROC IMPORT` and `PROC EXPORT` is due to the ability to access *named ranges* within Excel by using the `OLEDB` and `PCFILES` methods.

Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Excel on a 32-bit SAS Server Using OLEDB

You can read from a specific spreadsheet in an Excel Workbook by using this code:

```
libname olexls oledb provider='Microsoft.Jet.OLEDB.4.0'
                     properties=('Data Source'='C:\TEMP\Payors.xls')
                     provider_string='Excel 8.0;hdr=yes';
run;

proc print data=olexls.'PayorList$'n width=minimum;
run;

data PayorList;
 set olexls.'PayorList$'n;
run;

libname olexls clear;
run;
```

Notice that the name of the spreadsheet has a dollar sign placed at the end of it, is enclosed in tick marks, and has the letter-*n* at the end of the string. The letter-*n* indicates to SAS that the name is a non-standard SAS name. This is known as a SAS *name literal*. These are useful for representing names that do not follow the normal SAS naming conventions (such as Excel sheet names, database column names, database tables names, etc.).

Make sure to clear the `LIBNAME` to the Excel Workbook or you will not be able to open it up!

Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Excel on a 32-bit SAS Server Using OLEDB

You can use `PROC CONTENTS` to see the names of all of the sheets (as well as *named ranges*) within the workbook:

```
libname olexls oledb provider='Microsoft.Jet.OLEDB.4.0'
                     properties=('Data Source'='C:\TEMP\Payors.xls')
                     provider_string='Excel 8.0;hdr=yes';
run;

proc contents data=olexls._all_;
run;

libname olexls clear;
run;
```

```
Libref         OLEXLS
Engine         OLEDB
Physical Name  Microsoft.Jet.OLEDB.4.0
Schema/Owner   Admin

                          DBMS
                Member    Member
#  Name         Type      Type

1  BOB          DATA      TABLE
2  PayorList$   DATA      TABLE
3  Sheet2$      DATA      TABLE
4  Sheet3$      DATA      TABLE
```

Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Excel on a 32-bit SAS Server Using OLEDB

You can create a new Excel spreadsheet by specifying a non-existent XLS file on the properties line:

```
libname olexls oledb provider='Microsoft.Jet.OLEDB.4.0'
                     properties=('Data Source'='C:\TEMP\MyNewExcelWorkbook.xls')
                     provider_string='Excel 8.0;hdr=yes';
run;

data olexls.MyNewSheet;
 set PayorList; /* PayorList is a pre-existing SAS dataset */
run;

libname olexls clear;
run;
```

If your sheet name does not contain spaces or non-standard characters, you can forego the name literal syntax. Also, if your sheet name does contain spaces, SAS replaces them with underscores.

Also, note that you can populate the workbook with several sheets at once. That is, you don't always have to clear the `LIBNAME` and then re-assign it just to add another sheet to the workbook.

Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Excel on a 32-bit SAS Server Using OLEDB

```
libname olexls oledb provider='Microsoft.Jet.OLEDB.4.0'
                     properties=('Data Source'='C:\TEMP\MyNewExcelWorkbook.xls')
                     provider_string='Excel 8.0;hdr=yes';
run;

data olexls.MyNewSheet1;
 set PayorList; /* PayorList is a pre-existing SAS dataset */
run;

data olexls.MyNewSheet2;
 set PayerList; /* PayerList is a pre-existing SAS dataset */
run;

libname olexls clear;
run;
```

Take note of the `PROVIDER_STRING` option `HDR=`. When `HDR=YES`, SAS will name the dataset columns based on the first row in the sheet. If there is no header row, specify `HDR=NO` and SAS will name the dataset columns `F1`, `F2`, `F3`, …

Now, let's take a look at the `PROC CONTENTS` of `MyNewExcelWorkbook.xls`:

Reading from and Writing to Microsoft Excel and Access

sheepsqueezers.com

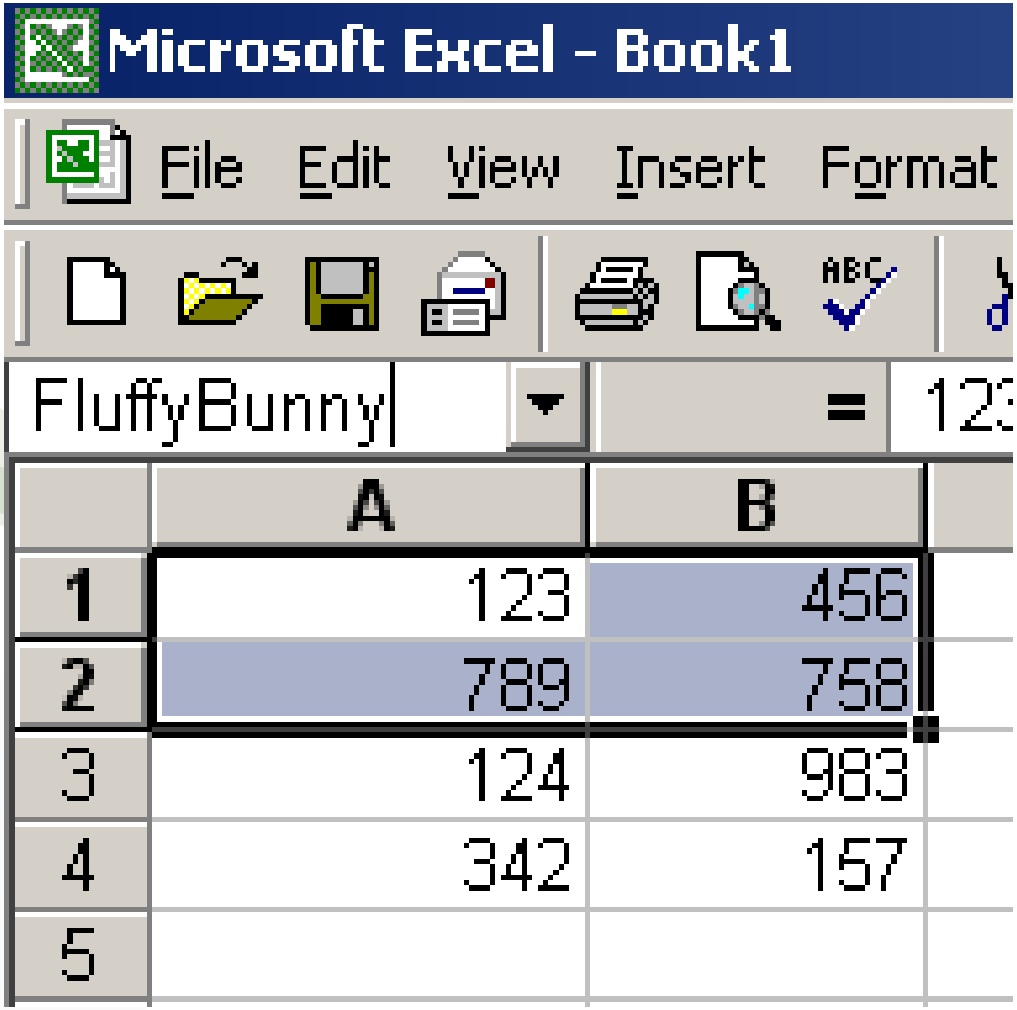
Reading from/Writing to Excel on a 32-bit SAS Server Using OLEDB

```
Libref          OLEXLS
Engine          OLEDB
Physical Name   Microsoft.Jet.OLEDB.4.0
Schema/Owner    Admin
                                  DBMS
                        Member    Member
   #  Name              Type      Type

   1  MyNewSheet1       DATA      TABLE
   2  MyNewSheet1$      DATA      TABLE
   3  MyNewSheet2       DATA      TABLE
   4  MyNewSheet2$      DATA      TABLE
```

You'll notice the when you create your own Excel workbook from within SAS, each sheet contains a sheet with the same name but with a dollar sign appended to it. As you see, there is a `MyNewSheet1` and a `MyNewSheet1$`. Before we talk about the difference, let's talk about named ranges in Excel.

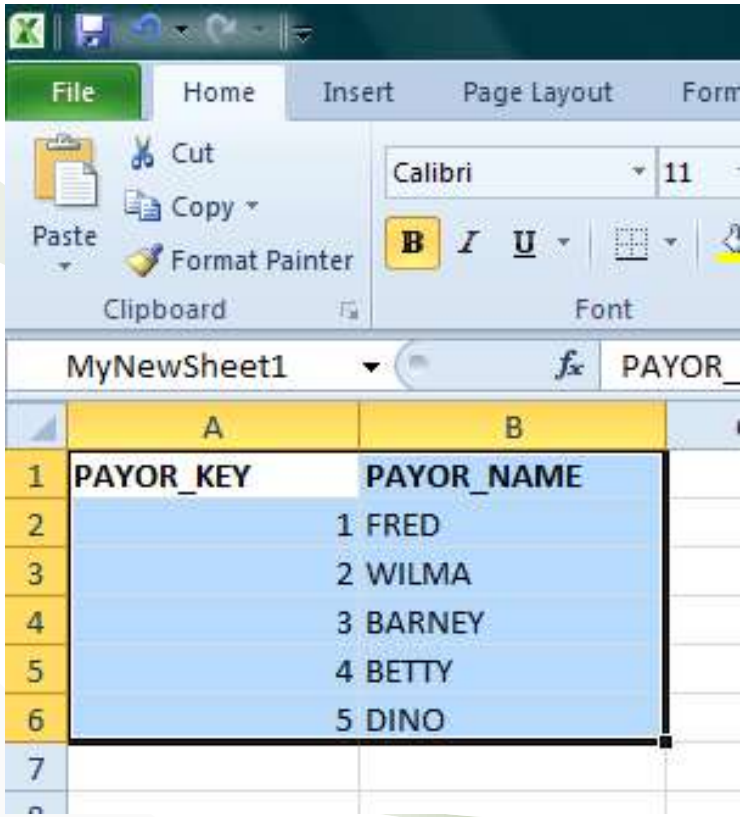
A *named range* is a highlighted set of rows and columns that are given a specific name. These names can be used in Excel formulas as a substitute for the typical row and column syntax: A1..A7. To create a named range within Excel, use your mouse to highlight a series of rows and columns, move your cursor to the name range input box(upper left corner below the File menu), and enter a name. On the next slide, we highlighted two rows and columns and named them *FluffyBunny*.

Reading from and Writing to Microsoft Excel and Access



Reading from/Writing to Excel on a 32-bit SAS Server Using OLEDB

Now, the sheet names *without the dollar signs* represent named ranges. The sheet names *with the dollar signs* refer to the actual sheet itself. We can test that theory by opening up our Excel workbook we just created from within SAS, and highlighting the entire set of data. Take note that the named range MyNewSheet1 appears in the named range input box:



Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Excel on a 32-bit SAS Server Using OLEDB

So, what's the big idea? Well, you cannot delete a "real" spreadsheet from an Excel workbook within SAS, but you can delete the data from the named range. This means that you can have a spreadsheet strewn with multiple named ranges throughout and can delete and replace the data at will. This includes data associated with charts and pivot tables!

```
libname olexls oledb provider='Microsoft.Jet.OLEDB.4.0'
                     properties=('Data Source'='C:\TEMP\MarketData.xls')
                     provider_string='Excel 8.0;hdr=yes';

run;

data DowData;
 set olexls.DowData;
run;

data NasdaqData;
 set olexls.NasdaqData;
run;

/* Bump the data up */
data DowData;
 set DowData;
 Close=Close+1000;
run;
```

Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Excel on a 32-bit SAS Server Using OLEDB

```
data NasdaqData;
 set NasdaqData;
 Close=Close+1000;
run;

/* Delete the data from the named ranges - NO SHEETS ARE REMOVED, JUST DATA IN RANGE!! */
proc datasets library=olexls;
 delete DowData NasdaqData;
run;
quit;

/* Insert the data back into the named ranges */
data olexls.DowData;
 set DowData;
run;

data olexls.NasdaqData;
 set NasdaqData;
run;

libname olexls clear;
run;
```

Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Access Databases on a 32-bit SAS Server Using OLEDB

You can read from a specific table in an Access database by using this code:

```
libname olemdb oledb provider="Microsoft.Jet.OLEDB.4.0"
 properties=('data source'="LOCATION-OF-ACCESS-DATABASE\ACCESS-DATABASE.mdb");
run;

data SAS-DATASET-NAME;
 set olemdb.'ACCESS-TABLE-NAME'n;
run;

libname olemdb clear;
run;
```

Notice that this is very similar to the Excel `LIBNAME`.

One major difference is that you MUST create a blank Access database (.mdb file) BEFORE you can use SAS to read from and write to a table.

Also, you can use PROC SQL as well in a similar fashion to the way we access Oracle and SQL Server. See the next slide:

Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Access Databases on a 32-bit SAS Server Using OLEDB

```
libname dbPROD01 oledb init_string="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
C:\MyAccessDB.mdb" INSERTBUFF=50000 DBCOMMIT=50000;
run;

proc sql noprint noerrorstop;
 connect to oledb as PROD01 (init_string="Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\MyAccessDB.mdb");

 execute(DROP TABLE TOP25) by PROD01;

 execute(CREATE TABLE TOP25(LOCALUNIVERSEKEY BYTE,
                            MNTH BYTE,
                            CLASS3 TEXT(255),
                            PCT SINGLE)) by PROD01;

 insert into dbPROD01.TOP25
  select LOCALUNIVERSEKEY,MNTH,CLASS3,PCT
   from work.TOP25;

 execute(CREATE INDEX IX_TOP25 ON TOP25(LOCALUNIVERSEKEY,MNTH)) by PROD01;

 disconnect from PROD01;
quit;
```

Reading from and Writing to Microsoft Excel and Access

sheepsqueezers.com

Reading from/Writing to Excel on a 64-bit SAS Server Using PCFILES

Since the Microsoft Jet OLEDB Provider is not available on 64-bit servers, the only way to read from/write to Excel is via *SAS/Access to PC Files* using the **PC File Server**. The PC File Server is a program always running on the 64-bit server waiting for you to contact it with the syntax below. Ensure the PC File Server software is running on your 64-bit SAS Server first!

You can read from a specific spreadsheet in an Excel Workbook by using this code:

```
libname XLSin pcfiles
        server="your_64bit_server_name"
        port="your_pc_files_port"
        path="\\networkUNC\Your-ExcelWorkbook.xls";
run;

data MyData;
 set XLSin.'MySheet$'n;
run;
```

Note that the `SERVER=` and the `PORT=` tell SAS how to connect to the PC File Server.

At this point, reading in a spreadsheet is the same as for the 32-bit machine:

As for the 64-bit machine, you can use Excel named ranges. Also, you can read from and write to Microsoft Access databases by replacing the path= above with something like this:

```
path="\\networkUNC\Your-AccessDatabase.mdb";
```

Reading from and Writing to Microsoft Excel and Access

Reading from/Writing to Excel on a 64-bit SAS Server Using PCFILES

Note that SAS will determine whether the `LIBNAME` points to an Excel workbook or Access database from the extension in the `PATH=` option. If your workbook or database file does not have a standard extension (`.xls` or `.mdb`), you can use either the `TYPE="EXCEL"` or `TYPE="ACCESS"` option to let SAS know which file type it:

```
libname MDBin pcfiles
        server="your_64bit_server_name"
        port="your_pc_files_port"
        type="access"
        path="\\networkUNC\Your-AccessDatabase.mdb";
run;
```

Also, when you are creating a new Excel workbook, you can let SAS know which version to create. Specify either `VERSION="2002"`, `VERSION="2000"` or `VERSION="97"`:

```
libname XLSin pcfiles
        server="your_64bit_server_name"
        port="your_pc_files_port"
        version="2002"
        type="excel"
        path="\\networkUNC\Your-ExcelWorkbook.xls";
run;
```

Creating a New SAS Function Using the "C" Language

Creating a New SAS Function Using the "C" Language

Please refer to the following document on the sheepsqueezers.com website:

Creating C DLLs for SAS

Useful SAS System Options

Useful 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, 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`
- `CMPOPT | NOCMPOPT` -- Specifies whether or not SAS optimizes the code generated. Specifying NOCMPOPT prevents compiler optimization, but more accurate error messages. Default: `CMPOPT`

Useful SAS System Options

- `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`
- `DATE | NODATE` – Specifies whether or not the date at which the SAS job began is printed in the log and listing files. DEFAULT: `NODATE`
- `DSNFERR | NODSNFERR` – Specifies what SAS will do if a SAS dataset is found or not. If `DSNFERR` is in effect, SAS prints an error message and then stops the program. If `NODSNFERR` is in effect, SAS will ignore the error message and continue processing.
- `DTRESET | NODTRESET` – Specifies whether or not the date is updated each time a page is written to the SAS log or listing files. Normally, SAS only prints the start time of the SAS job and does NOT update the time as the program continues to run. Default: `NODTRESET`
- `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`.
- `FMTSEARCH=(cat1,cat2,…)` – Specifies the order in which format catalogs are searched. For example, `FMTSEARCH=(WORK.FORMATS,LIBRARY.FORMATS)` are always searched first if not specified, unless they appear in the list.
- `FORMCHAR="formatting-chars"` – Specified the formatting characters used for printed output for procedures like `FREQ`, `TABULATE`, etc. Default depends on the system, but recommend: `FORMCHAR="|----|+|---+=|-/\<>*"`.

Useful SAS System Options

- `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`
- `MSGLEVEL = N|I` – Specifies whether notes/warnings/errors are only printed (when `MSGLEVEL=N`) or whether much more detailed information is printed to the SAS log (`MSGLEVEL=I`). When `MSGLEVEL=I` is set, SAS prints additional information about index usage, merge processing, and more. Default: `MSGLEVEL=N`, but author recommends `MSGLEVEL=I`
- `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.
- `REPLACE | NOREPLACE` – Specifies whether or not permanently stored can be replaced during SAS processing. Has no affect on WORK datasets. Default: REPLACE.

Useful SAS System Options

- SASAUTOS=(lib1, lib2, …) – Specifies the autocall macro library search path. Normally, lib1 is set to SASAUTOS in order for programmers to use the SAS-supplied macros.
- SKIP = # – Specifies the number of lines to skip at the top of SAS output.
- SORTDUP = PHYSICAL | LOGICAL – Specifies whether the PROC SORT procedure removes duplicate records – when you use the NODUP or NODUPREC option, of course – based on ALL of the variables in the dataset being sorted (when SORTDUP=PHYSICAL) or whether PROC SORT removes duplicate rows based on the variables remaining AFTER all DROP= and KEEP= dataset options are taken into account (when SORTDUP=LOGICAL). Has no affect if no DROP= and KEEP= dataset options are specified.
- SORTEQUALS | NOSORTEQUALS – Specifies whether observations with identical BY variable values in a PROC SORT are to retain their relative position in the output dataset as they are in the input dataset (when SORTEQUALS is specified). If NOSORTEQUALS is specified, observations with the same BY variable values are sorted in no particular order. SAS recommends NOSORTEQUALS to achieve best performance. Please see the PROC SORT options EQUALS and NOEQUALS, which override the SAS System Option SORTEQUALS and NOSORTEQUALS.
- 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.

Useful SAS System Options

- `MPRINTNEST | NOMPRINTNEST` – Specifies whether or not SAS will display nesting of SAS macros in the `MPRINT` output.

Useful SAS Dataset Options

Useful 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
- `GENMAX=#` – Specifies how many generations of a SAS dataset you want to keep. Default: 1. Previous datasets are named `dataset-name#001`, etc. Note that the maximum SAS dataset name size is 28 characters, not 32. Maximum of 1000 generation datasets saved.

Useful SAS Dataset Options

- `GENNUM=#` – Specifies which generation you want to use. If you specify a positive number, it is an absolute reference. If you specify a negative number, it is a relative reference back from the most recent dataset. For example, if you specify `SET BOB(GENNUM=3)` then `BOB#003` is used. If you specify `SET BOB(GENNUM=-1)` then the previous dataset is used.
- `IDXNAME=`*`index-name`* – Specifies which index to use in an indexed dataset during WHERE clause processing. Normally, SAS determines the best index, but you can override this setting with this option.
- `IDXWHERE=YES|NO` – Specifies whether or not SAS uses an index at all on an indexed dataset access by WHERE clause processing. If you specify NO then SAS performs sequential processing.
- `IN=`*`variable`* – Creates a variable named *`variable`* that indicates whether the dataset contributed data to the current observation.
- `INDEX=(index-spec-1 … index-spec-N)` – Creates one or more indexes on your SAS dataset. Index specification can the name of a variable in which case the index created in called a **simple index**; or, the index specification can be in the form of *`index-name=(variable-1 variable-2 …)`* and is known as a **composite index**. A simple index creates an index on only one variable, whereas a composite index creates an index on all specified variables in combination, and not individually. Example: `data wilma(index=PATIENT_KEY);` Example: `data dino(index=(idxPATDOC=(PATIENT_KEY PRACTITIONER_KEY)));` You can also provided both simple and composite indexes:

```
data bambam(index=( AGE_KEY idxPATNDC=(PATIENT_KEY NDC_KEY) ) );
```

Useful 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`
- `OUTREP=`*`output-format`* – Specifies the data representation for the SAS dataset. *`Output-format`* can be `WINDOWS, WINDOWS_64, VAX_VMS, LINUX,` etc. See documentation for more output formats.
- `POINTOBS=YES|NO` – Specifies whether or not a compressed SAS dataset can be accessed via the `POINT=` dataset option. Default: `POINTOBS=YES`. Specifying `POINTOBS=NO` will improve performance by roughly 10% when updating or adding to a compressed SAS dataset.
- `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.
- `REUSE=NO|YES` – Specifies whether or not new observations are written to the free space within a compressed SAS dataset. If NO, SAS appends to the dataset. For most things we do, set `REUSE=NO`.
- `ROLE=FACT|DIMENSION` – Allows you to specify which dataset is the FACT table in a SQL join. If you specify `ROLE=FACT` on a specific dataset, the remaining are considered `ROLE=DIMENSION`. If you specify `ROLE=DIMENSION` on all tables but one, the remaining table is considered `ROLE=FACT`. These dataset options help improve join performance. These options are in effect only during the join.

Useful SAS Dataset Options

- `SORTEDBY=sorted-var-1 sorted-var-2…` – Use this option to let SAS know is 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.

PROC-ology – New Options for Select Procedures

PROC-ology – New Options for Select Procedures

In this section, we take a look at the most common SAS procedures and outline some of the new statements and options you might not be aware of.

PROC SORT

- The DUPOUT=*dataset-name* option allows you to capture duplicate observations removed by using the options NODUPKEY or NODUPREC. For example,

```
proc sort data=MyDataset
        out=MyDeDupedDataset
        dupout=MyDups nodupkey;
 by var1 var2 var3;
run;
```

- EQUALS | NOEQUALS – SAS normally we maintain the order of the observations from the incoming dataset when creating the output dataset if there are multiple records for the BY variables. That is, EQUALS is the default. If you do not need to maintain the order of the observations, use the NOEQUALS option. According to the SAS manual, using the NOEQUALS option can save CPU time and memory.
- OVERWRITE – Specifying this option tells PROC SORT to delete the incoming dataset before the sorted output dataset is fully created. This can save disk space.

PROC-ology – New Options for Select Procedures

PROC MEANS

- The `TYPES` statement allows you to specify which combination of the variables named on the `CLASS` statement are to be created in the `OUTPUT OUT=` dataset. For example, the following `PROC MEANS` will only output `_TYPE_='110'b (6)` and `_TYPE_='101'b (5)`:

```
proc means data=MyDataset noprint;
 class var1 var2 var3;
 var wilma;
 types var1*(var2 var3); /* equivalent to var1*var2 var1*var3 */
 output out=bob sum(wilma)=BigWilma;
run;
```

- The `WAYS` **statement** allows you to specify whether you want all two-way, three-way, etc. combinations in the output dataset. Use this instead of `TYPES`. For example:

```
proc means data=MyDataset noprint;
 class var1 var2 var3;
 var wilma;
 ways 2 3; /* _TYPES_ is 3,5,6,7 */
 output out=bob sum(wilma)=BigWilma;
run;
```

PROC-ology – New Options for Select Procedures

PROC MEANS

- The `OUTPUT` statement allows you to specify several options. The `LEVELS` option creates a variable named `_LEVEL_` in the output dataset which indicates the row number within each `_TYPE_` value:

```
proc means data=MyDataset noprint;
 class var1 var2 var3;
 var wilma;
 types var1*(var2 var3);
 output out=bob sum(wilma)=BigWilma/levels ways;
run;
```

- The `WAYS` **option** creates a variable named `_WAY_` in the output dataset which indicates the number of variables used in the calculation.

See next slide for example output.

PROC-ology – New Options for Select Procedures

Obs	var1	var2	var3	_WAY_	_TYPE_	_LEVEL_	_FREQ_	BigWilma
1	.	1	3	2	3	1	3	2.12433
2	.	2	3	2	3	2	2	0.65922
3	.	3	3	2	3	3	1	0.92160
4	1	.	3	2	5	1	1	0.18496
5	2	.	3	2	5	2	1	0.97009
6	3	.	3	2	5	3	1	0.39982
7	4	.	3	2	5	4	1	0.25940
8	5	.	3	2	5	5	2	1.89088
9	1	1	.	2	6	1	1	0.18496
10	2	1	.	2	6	2	1	0.97009
11	3	2	.	2	6	3	1	0.39982
12	4	2	.	2	6	4	1	0.25940
13	5	1	.	2	6	5	1	0.96928
14	5	3	.	2	6	6	1	0.92160
15	1	1	3	3	7	1	1	0.18496
16	2	1	3	3	7	2	1	0.97009
17	3	2	3	3	7	3	1	0.39982
18	4	2	3	3	7	4	1	0.25940
19	5	1	3	3	7	5	1	0.96928
20	5	3	3	3	7	6	1	0.92160

PROC-ology – New Options for Select Procedures

PROC MEANS

- The PROC MEANS option COMPLETETYPES tells PROC MEANS to create all combination of the CLASS variables even if that combination does NOT exist in the incoming dataset. Notice that A=2 and B=2 does NOT occur.

Obs	a	b	value
1	1	1	0.18496
2	1	2	0.97009
3	2	1	0.39982

```
proc means data=MySmallDataset noprint completetypes;
 class a b;
 var value;
 output out=SumData sum(value)=value/levels ways;
run;
```

Obs	a	b	_WAY_	_TYPE_	_LEVEL_	_FREQ_	value
1	.	.	0	0	1	3	1.55488
2	.	1	1	1	1	2	0.58479
3	.	2	1	1	2	1	0.97009
4	1	.	1	2	1	2	1.15505
5	2	.	1	2	2	1	0.39982
6	1	1	2	3	1	1	0.18496
7	1	2	2	3	2	1	0.97009
8	2	1	2	3	3	1	0.39982
9	2	2	2	3	4	0	.

PROC-ology – New Options for Select Procedures

PROC MEANS

- The PROC MEANS option DESCENDTYPES tells PROC MEANS to sort the output dataset by descending _TYPE_ instead of the normal ascending _TYPE_. This means that you will have _TYPE_=0 appear as the **last** row in the dataset:

```
proc means data=MySmallDataset noprint completetypes descendtypes;
 class a b;
 var value;
 output out=SumData sum(value)=value/levels ways;
run;
```

Obs	a	b	_WAY_	_TYPE_	_LEVEL_	_FREQ_	value
1	1	1	2	3	1	1	0.18496
2	1	2	2	3	2	1	0.97009
3	2	1	2	3	3	1	0.39982
4	2	2	2	3	4	0	.
5	1	.	1	2	1	2	1.15505
6	2	.	1	2	2	1	0.39982
7	.	1	1	1	1	2	0.58479
8	.	2	1	1	2	1	0.97009
9	.	.	0	0	1	3	1.55488

Naturally, _WAY_ and _LEVEL_ will follow suit.

PROC-ology – New Options for Select Procedures

PROC MEANS

- The `OUTPUT` statement allows you to specify one or more `IDGROUP` options which lets you to keep extreme observations from the incoming dataset, but they are placed in the output dataset as additional columns. For example, here is our input dataset:

Obs	NDC_KEY	PATIENT_KEY	COPAY_AMT
1	11111111111	1	10
2	11111111111	2	20
3	11111111111	3	30
4	11111111111	4	40
5	22222222222	1	60
6	22222222222	2	70
7	22222222222	3	80
8	22222222222	4	90

```
proc means data=NDC_COPAY noprint nway;
 class NDC_KEY;
 var COPAY_AMT;
 output out=AvgCoPay mean(COPAY_AMT)=AvgCoPayAmt
        idgroup(min(COPAY_AMT) obs out[2](COPAY_AMT)=MinCP)
        idgroup(max(COPAY_AMT) obs out[2](COPAY_AMT)=MaxCP);
run;
```

NDC_KEY	AvgCoPayAmt	MinCP_1	MinCP_2	_OBS_1	_OBS_2	MaxCP_1	MaxCP_2	_OBS2_1	_OBS2_2
11111111111	25	10	20	1	2	40	30	4	3
22222222222	75	60	70	5	6	90	80	8	7

PROC-ology – New Options for Select Procedures

PROC FREQ

- The `TABLES` statement allows you to specify the `SPARSE` option if you want all combinations of variable values to be created. This is similar to `COMPLETETYPES` for `PROC MEANS`. For example,

```
data MySmallDataset;
 a=1;b=1;value=uniform(1);output;
 a=1;b=2;value=uniform(1);output;
 a=2;b=1;value=uniform(1);output;
run;

proc freq data=MySmallDataset;
 tables a*b/list sparse;  /* can use noprint and out=dataset-name options as well */
run;
```

a	b	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	1	1	33.33	1	33.33
1	2	1	33.33	2	66.67
2	1	1	33.33	3	100.00
2	**2**	**0**	**0.00**	**3**	**100.00**

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`.