

Unleash the Power of Less Well Known but Useful SAS® DATA Step Functions

Timothy J. Harrington, Navitas, Inc.

ABSTRACT

SAS Version 9.4 has over 400 documented SAS functions and procedures. Some SAS users may have grown accustomed to using only a few of the more widely and well-known of these, and have had to invest significant time and extra effort manipulating code to achieve a desired result. This paper describes a selection of these SAS functions and how they can be used to obtain specific results with minimal coding. Solutions to a variety of programming challenges are demonstrated and the advantages of these techniques are discussed and compared to more traditional means. The functions included here are available in SAS v9.4, some may not be available in earlier versions.

INTRODUCTION

The functions and facilities listed are some of the more common and generic that are available in recent versions of SAS. Examples of both code and listed output are shown to demonstrate their use and versatility. When discussing each function, or related group of functions, the ease of coding and run time performance are compared with more traditional methods. There are also short macros listed which make use of these functions and can be applied to common specific programming situations. Due to space limitations and sometimes only very specific and uncommon applications, only the functions considered most useful for common programming situations are included.

NUMERICS

FACTS ABOUT COLLECTIONS OF NUMBERS

There are a large number of functions in SAS which return a result about a group of numeric values (constants, variables, or successive array elements). Common examples are:

```
total=sum(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10);
average=mean(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10);
midpoint=median(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10);
highest=max(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10);
lowest=min(x1,x2,x3,x4,x5,x6,x7,x8,x9,x10);
```

A point worth noting here is the variables as listed above could be expressed as array elements, for example if x1 through x10 are defined as an array named *sample*:

```
array sample {10} x1-x10;
```

the total above could be called as

```
total=sum(of sample{*});
```

or as

```
total=sum(of x1-x10);
```

Note: if 'of' is not specified before a range the dash is interpreted as a minus sign and the argument is the difference between the two numbers, in this case x1 minus x10

Another important factor is how missing values are handled. In SAS missing values are considered as 'less than' negative numbers, hence, for example, the 'minimum' of {.,-15,-2,0,7,24,49} would be the missing value (.), however the MIN function ignores missing values, hence in this example the returned value would be -15. In other functions, such as SUM, missing is taken as zero.

A need to know maximum and minimum values is a frequent requirement, but occasionally there is a need to find out the second smallest or second largest number. SAS has two functions LARGEST and SMALLEST, which return the *n*th largest or *n*th smallest number. The first argument to either of these functions is *n*. For example, if X1=9, X2=44, X3=12, X4=53, X5=59, X6=83, X7=11, X8=73, X9=74, and X10=43

```
largest(2,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)
```

returns the second largest of the numbers x1 through x10, which is 74 (X9) (Largest is X6(83))

```
smallest(3,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10)
```

returns the third smallest, which is 12 (X3) (Smaller values are X1 (9) and X7 (11))

If there are multiple occurrences of the same number, each occurrence is counted separately as though the numbers were sorted in ascending or descending sequence. For example, if X1=79, X2=85, X3=24, X4=24, X5=31, X6=31, X7=24, X8=17, X9=52, and X10=90 and the LARGEST function is used, when *n*=1 the result is 90, *n*=2 results in 85, *n*=3 results in 79, *n*=4 results in 52, *n*=5 results in 31, *n*=6 also returns 31, *n*=7 returns 24, *n*=8 returns 24, and *n*=9 also returns 24, and *n*=10 returns 17. If *n* is zero, negative, missing, or greater than the number of numbers being tested the returned result is missing.

These two functions only work with non-missing values, if missing values are also being considered the ORD (ordinal) function should be used. The syntax for this is exactly the same as for SMALLEST except missing values (. and .a through .z) may be included.

When there is a need to sort numbers in ascending sequence there is SORTN. The numbers may be the elements of an array. For example, using the above array *sample* with the above values of X1 through X10:

```
call sortn(of sample{*})
```

would sort the values such that X1=17, X2=24, X3=24, X4=24, X5=31, X6=31, X7=52, X8=79, X9=85, X10=90. SORTN also handles missing values, counting them as 'less than' negative numbers.

Whilst discussing missing values, the number of missing values in a list is given by the NMISS function (or CMISS for character arguments), if there are no missing values present the returned result is 0.

Two other useful mathematical functions are LCM and GCD, Lowest Common Multiple and Greatest Common Denominator, respectively. For example, if, in a clinical study a patient takes two medications A and B, and A is given every *a_freq* days and B every *b_freq* days the frequency of both medications being taken at the same time, *ab_freq*, is:

```
ab_freq=lcm(a_freq,b_freq);
```

For example, if *a_freq* is 17 days and *b_freq* is 19 days *ab_freq* is once every 323 days.

Now, supposing there is a need to determine a cycle length to match multiple dosing regimens. The cycle needs to be the GCD of all of the regimens, so that any dose of any of the medications is at the end of a cycle. (Some cycles may not have any doses) If there are four medications with dosing frequencies of *a_freq*, *b_freq*, *c_freq*, and *d_freq*, the largest applicable cycle length, *abcd_cyc* would be

```
abcd_cyc=gcd(a_freq,b_freq,c_freq,d_freq);
```

If *a_freq*=24 days, *b_freq*=30 days, *c_freq*=42 days, and *d_freq*=60 days, *abcd_cyc* is 6 days

If there is no common denominator, for example, the numbers 24,35, and 60, or if any of the numbers is a prime number, the returned result is missing.

Finally, a quick look at array functions. In SAS there are DIM, HBOUND, and LBOUND. All of these take the array name as an argument and an array dimension as an optional second argument. DIM returns the number of elements in the array, whether starting with 1 or otherwise. For example

```
x=dim(sample);
```

returns the number of elements (10) in the array *sample*, defined above. For a two dimensional array, *sample2*:

```
x=dim{sample2,1};
y=dim(sample2,2);
```

x is the number of elements in the first dimension and Y is the number of elements in the second dimension. DIM is a useful function to ensure an attempt is not made to access an element out of range.

HBOUND is similar to DIM except it returns the absolute upper bound instead of just the array dimension size. In the examples above if HBOUND had been used instead of DIM, x and y would have had the same values. However, if *sample2* had been defined as an array with its first dimension starting at, say, 5 and ending at 15, DIM would still return 10 because there are still 10 elements, but HBOUND would return 15. Similarly, LBOUND is the lowest element number and would have the value 5. If no starting element number is specified LBOUND is 1.

MAKING SELECTIONS

There are several ways in which values may be assigned differently based on conditional code. Examples are *if then else*, *select*, *sql case.. when ... then ...end* to name but a few. When conditional execution depends on the result of a Boolean (true or false) numeric result a convenient function is IFN. The syntax of the IFN call is:

```
ifn(<condition>,<value when true>,<value when false>,<value when missing>);
```

When the first argument (the condition) is true IFN returns the value of the second argument (when true), when the first argument is not true IFN returns the value of the third argument (when false). The fourth argument is optional, and, if specified, is the returned value when missing. In this example, r is 1 when $x=y$ and r is -1 when x is not equal to y :

```
r=ifn(x=y,1,-1);
```

If x or y but not both were missing r would be -1 because x is not equal to y . If both x and y were missing (.) x is equal to y so r would be 1. If different missing values, such as *.a* and *.b* were specified r would be -1. In this case the result is never missing since the expression $x=y$ will always have a non-missing result (even though SAS propagates missing values in arithmetic expressions).

In this example, however:

```
r=ifn(x,1,-1,0);
```

when x is 1 r will be 1, when x is 0 r will be -1, but when x is missing r will be 0.

In this next example a character string *result* is assigned a value 'Pass' or 'Fail', based on the numeric value of *score*. The 'pass mark' is 40. When *score* is 40 or more IFN returns 1 and when *score* is 39 or less IFN returns 2. The 1 or the 2 is then used by the SCAN function to select the first word 'Pass' or the second word 'Fail'.

```
result=scan('Pass Fail',ifn(score>=40,1,2));
```

More complex situations can be handled by nesting calls of IFN, for example:

```
result=scan('Distinction Merit Pass Fail',
  ifn(score>=85,1,ifn(score>=70,2,ifn(score>=40,3,4))));
```

Here, the innermost IFN call determines if *score* is 40 or more, the next level determines if *score* is 70 or more, the outermost level determines if *score* is 85 or more. If *score* is less than 40 the innermost call returns the 'false' option (4) and since the two higher calls of IFN are both 'false' in this case the net result is the 4, which maps to 'Fail'. If *score* is 40 or more but less than 70 the innermost call returns the 'true' value (3), this becomes the 'false' value for the next level and, since *score* is less than 70 the 'false' value of 3 is returned, this result is subsequently 'false' for the outermost call, resulting in 'Pass'. If *score* is 70 or more but less than 85 the second level IFN return is 'true' (2) but the outermost is false, so the 2 is returned overall which maps to 'Merit'. If *score* is 85 or higher the outermost call is true so the result is 1, which maps to 'Distinction'.

CHARACTER STRING INFORMATION AND MANIPULATION

Most SAS Programmers are familiar with string functions such as SUBSTR, and UPCASE/LOWCASE. Whilst these SAS functions are invaluable tools there are some situations where a more tailor-made facility would be easier to use. There are a large number of string related SAS functions, included below are the most common, and in practice, the most helpful. Using the items listed below, code can be written such that problems caused by unexpected data values, missing values, and implicit type conversion are prevented.

TO BE OR NOT TO BE – ANY AND NOT

There are many times when a programmer needs to know what is at a given position in a character string, or where a particular character type or attribute is present (or absent), such as upper or lower case, numeric, punctuation, or a space. To achieve this, there are the following seven ANY functions and seven corresponding NOT functions.

SAS ANY/NOT Function Name	Finds the position of the first (or next) occurrence of
ANYALNUM	Any alphanumeric character ('A' to 'Z', 'a' to 'z', or '0' to '9')
ANYALPHA	Any alphabetic character ('A' to 'Z' or 'a' to 'z')
ANYUPPER	Any upper case character ('A' to 'Z')
ANYLOWER	Any lower case character ('a' to 'z')
ANYDIGIT	Any character which is a numeric digit ('0' to '9')
ANYPUNCT	Any 'punctuation' character such as { . , ' " ! ? ; : () }
ANYSpace	Any blank, including leading and trailing blanks
NOTALNUM	Any character which is not alphanumeric (including spaces)
NOTALPHA	Any non-alphabetic character
NOTUPPER	Any non-alphabetic or lower case character
NOTLOWER	Any non-alphabetic or upper case character
NOTDIGIT	Any character which is not a numeric digit ('0' to '9')
NOTPUNCT	Any character which is not a 'punctuation' symbol
NOTSPACE	Any non-blank character

In each case there are two arguments, the character string, and the numeric position within the string to begin the search, the default start position is 1, the first character within the string (including any leading spaces). The return value is the position of the first occurrence within the string relative to the starting position specified in the second argument. When the search finds no applicable occurrences a zero is returned. A zero value is also returned if the second argument is zero, missing, negative, or greater than the declared length of the string. Leading and trailing spaces are also considered. (LEFT removes leading spaces, TRIM removes trailing spaces until the text is stored in another variable)

Here are some examples of the ANY functions using the following character string HEADING:

HEADING="Calling all SAS Programmers to Pharmsug 2019! Learning (& fun) for all in Philadelphia PA on June 15/16-19 2019."

In each case below, the function returns the result, *location*, which is the position in HEADING where the first occurrence of a character satisfying the function is found. The value of *text*, is the character located at that point, i.e.: substr(heading,location,1).

Function Call	Location	Text	How the Result is Found
ANYALNUM(heading,1)	1	'C'	First alphanumeric character is the 'C' in 'Calling'
ANYALPHA(heading,1)	1	'C'	First alphabetic character is the 'C' in 'Calling'
ANYUPPER(heading,1)	1	'C'	First upper case alphabetic character is the 'C' in 'Calling'
ANYLOWER(heading,1)	2	'a'	First lower case alphabetic character is the 'a' in 'Calling'
ANYDIGIT(heading,1)	42	'2'	First numeric character is the '2' in the first '2019'
ANYPUNCT(heading,1)	46	'!'	First punctuation symbol is the '!' after '2019'
ANYSPACE(heading,1)	8	' '	First space is between 'Calling' and 'all')
ANYALNUM(heading,25)	25	'e'	First alphanumeric from position 25 is the 'e' in 'Programmers')
ANYALPHA(heading,59)	60	'f'	First alphabetic character from position 59 is the 'f' in 'fun'
ANYUPPER(heading,33):	48	'L'	First upper case letter from position 33 is the 'L' in 'Learning'
ANYLOWER(heading,95)	96	'u'	First lower case letter from position 95 is the 'u' in 'June'
ANYDIGIT(heading,80)	100	'1'	First numeric digit from position 80 is the '1' in 'June 15'
ANYPUNCT(heading,63)	63	')'	First symbol from position 63 is the closing parenthesis of '(& fun)'
ANYSPACE(heading,48)	56	' '	First space from position 48 is the space between 'Learning' and the opening parenthesis

The ANY functions can be nested, that is the result of an inner call plus one be used as the starting point for an outer call. These examples return the location of the second occurrence of a character satisfying the function.

Function Call	Location	Text	How the Result is Found
ANYALNUM(heading,ANYALNUM(heading,1)+1)	2	'a'	Second alphanumeric character is the 'a' in 'Calling'
ANYALPHA(heading,ANYALPHA(heading,1)+1)	2	'a'	Second alphabetic character is the 'a' in 'Calling'
ANYUPPER(heading,ANYUPPER(heading,1)+1)	13	'S'	Second upper case character is the first 'S' in 'SAS'
ANYLOWER(heading,ANYLOWER(heading,1)+1)	3	'l'	Second lower case character is the first 'l' in 'Calling'
ANYDIGIT(heading,ANYDIGIT(heading,1)+1):	43	'0'	Second numeric character is the '0' in '2019'

Function Call	Location	Text	How the Result is Found
ANYPUNCT(heading,ANYPUNCT(heading,1)+1)	57	'('	Second punctuation symbol is the opening parenthesis of '& fun'
ANYSPACE(heading,ANYSPACE(heading,1)+1)	12	' '	Second space is between 'all' and 'SAS'
ANYLOWER(heading,ANYPUNCT(heading,1)+1)	49	'e'	The 'e' in 'Learning' is the first lower case letter after the first punctuation symbol, the '!'.

The NOT functions are the 'opposite' of the corresponding ANY functions, they find the location of the first (or next) occurrence of a character which is not of the specified type. NOTALPHA, for example, would return the first character which is not 'A' to 'Z' and not 'a' to 'z', that is a numeric digit, punctuation symbol, or space. Leading and trailing spaces are counted, LEFT, TRIM, and STRIP may be used to exclude reference to them.

Function Call	Location	Text	How the Result is Found
NOTALNUM(heading,1)	8	' '	First non-alphanumeric character is the space after 'Calling'
NOTALPHA(heading,5)	8	' '	First non-alphabetic character after the 'i' in 'Calling' is the space after 'Calling'
NOTUPPER(heading,101)	101	'5'	First non-uppercase character from position 101 is the '5' in 'June 15'
NOTLOWER(heading,100)	100	'1'	First non-lowercase character from position 101 is the '1' in 'June 15'
NOTDIGIT(heading,21)	21	'r'	First non-numeric character from position 21 is the second 'r' in 'Programmers'
NOTPUNCT(heading,72)	72	' '	First non-punctuation character from position 72 is the space between 'all' and 'in'.
NOTSPACE(heading,20)	20	'g '	First non-space character from position 20 is the 'g' in 'Programmers'
NOTSPACE(heading,ANYPUNCT(heading,1)+1)	48	'L'	First non-blank character after the first punctuation character is the 'L' in 'Learning' following the '!'.

Considering a more detailed example, the code below counts the number of each of the character types in the string HEADING as well as the total number of non-alphanumeric and non-blank characters. The variables CAPS and PUNCSYMS are character strings which will be used to contain a list of the distinct upper case letters and punctuation symbols. The FIND function is used to check CAPS or PUNCSYMS to see if the current character (C) from HEADING has been previously added to the string. FIND is like the

INDEX function except only one character (the second argument) is searched for in the first argument. If the character is found in the string being searched FIND returns a 1, otherwise a 0.

```

Length c $1 caps puncsyms $200;
n_alpnum=0; /* Initialize counters to zero and result strings to blank*/
n_alpha=0;
n_caps=0;
n_lower=0;
n_digit=0;
n_punct=0;
n_blank=0;
nonalnum=0;
nonblank=0;
caps=' '; /* Must be of sufficient length */
puncsyms=' ';

do i=1 to length(heading); /* Parse the string (not trailing blanks) */
  c=substr(heading,i,1);
  n_alpnum=n_alpnum+(anyalnum(c)=1);
  n_alpha=n_alpha+(anyalpha(c)=1);
  n_caps=n_caps+(anyupper(c)=1);
  n_lower=n_lower+(anylower(c)=1);
  n_digit=n_digit+(anydigit(c)=1);
  n_punct=n_punct+(anypunct(c)=1);
  n_blank=n_blank+(anyspace(c)=1);
  nonalnum=nonalnum+(notalnum(c)=1);
  nonblank=nonblank+(notspace(c)=1);
  if anyupper(c)=1 then do;
    if find(caps,c)=0 then do;
      caps=trim(left(caps)||c);
    end;
  end;
  if anypunct(c)=1 then do;
    if find(puncsyms,c)=0 then do;
      puncsyms=trim(left(puncsyms)||c);
    end;
  end;
end;

put / heading= ;
put n_alpnum= n_alpha= n_caps= n_lower= n_digit= n_punct= n_blank=
  nonalnum= nonblank=;
put caps=;
put puncsyms=;

```

HEADING=Calling all SAS Programmers to Pharnasug 2019! Learning (& fun) for all in Philadelphia PA on June 15/16-19 2019.

N_ALPNUM=88 N_ALPHA=74 N_CAPS=11 N_LOWER=63 N_DIGIT=14 N_PUNCT=7 N_BLANK=18
NONALNUM=25 NONBLANK=95

CAPS=CSAPLJ

PUNCSYMS=!(&)/-.

Taking the use of these functions a stage further is the macro listed below, ANYNOTFN. This macro is called from within a SAS DATA step and returns the position of a user specified occurrence from the

beginning or end of a character string and the character found at that point. This code allows any of the functions listed above to be used recursively the specified number of times, from either the start or the end of a text string. The parameters are *func*, the function name, *str*, the text string to be searched, *occ*, the occurrence number, *fpos*, the returned position in *str* of the occurrence found, and *ftext*, is the character found at that point. If the *occ* value is a positive integer the search is performed from the start of the string, or if *occ* is a negative integer the search is performed from the end of the string. If the specified occurrence is not found in *str*, *fpos* is returned as zero and *ftext* is blank. When searching from the end of the text the REVERSE function is used to reverse the text and the search is performed from the beginning of the backward text. The TRIM function is needed to remove the trailing spaces when the length of *str1* is less than its declared length. When the occurrence is found *fpos* is calculated by subtracting that point from the length of *str1* and adding one.

```
%macro anynotfn(func=,str=,occ=1,fpos=,ftext=);
  &fpos=.;
  &ftext=' ';
  %if &occ>0 %then %do; /* Start at the beginning of the text */
    &fpos=&func(&str, %do i=1 %to &occ; &func(&str, %end; 1 %do i=1 %to
      &occ; %if &i>=2 %then %do; +1 %end; ) %end; );
  %end;
  %else %if &occ<0 %then %do; /* Start at the end of the text */
    %let occ=%eval(&occ*-1);
    &fpos=&func(reverse(trim(&str)),
      %do i=1 %to &occ; &func(reverse(trim(&str)), %end; 1 %do i=1 %to
        &occ; %if &i>=2 %then %do; +1 %end; ) %end; );
    &fpos=length(&str)-&fpos+1;
  %end;
  if 0<&fpos<=length(&str) then do;
    &ftext=substr(&str,&fpos,1);
  end;
%mend anynotfn;
```

Example Macro Call	<i>fpos</i>	text	Result (<i>ftext</i>)
%anynotfn(func=anypunct, str=heading, occ=1, fpos=pos, ftext=text);	46	'!	First punctuation character, the '!' after '2019'
%anynotfn(func=anypunct, str=heading, occ=5, fpos=pos, ftext=text);	102	'/'	Fifth punctuation character, the '/' in 'June 15/19'
%anynotfn(func=anypunct, str=heading, occ=8, fpos=pos, ftext=text);	0		Only seven punctuation characters are present in the string
%anynotfn(func=anypunct, str=heading, occ=-1, fpos=pos, ftext=text);	113	'.'	Last punctuation character
%anynotfn(func=anydigit, str=heading, occ=-7, fpos=pos, ftext=text);	104	'6'	Seventh numeric digit from the end of the string, the '6' in 'June 15/16-19'
%anynotfn(func=notlower, str=compress(heading), occ=5, fpos=pos, ftext=text);	14	'P'	The COMPRESS function removes all the spaces, after this the fifth non-lower case character is the 'P' in 'Programmers'.

MAKING SELECTIONS

When there is a need to select a text string based on a numeric assignment there is CHOOSEC. This takes a positive numeric integer to return the corresponding text item. In the example below *aegraden* is a numeric integer in the range 1 to 5. When *aegraden* is 1 the first string ('Mild') is returned in the character string *ae_grade*, when *aegraden* is 2, the result is 'Moderate' and so on. If *aegraden* is less than 1 or greater than the total number of argument strings *ae_grade* is returned as blank and an 'Invalid Argument' note is written to the saslog. Care must be taken to ensure the declared length of the result string is at least as long as the longest CHOOSEC argument or truncation will occur.

```
ae_grade=choosec(aegraden, 'Mild', 'Moderate', 'Severe',  
'Life Threatening', 'Fatal');
```

Another very useful selection function is IFC. This replaces the need for 'if ... then ... else... do ... end' code. IFC, like its numeric sibling IFN, takes three arguments, the condition being tested, the text to use when the condition is true, the text to use when the condition is false, and optional text to use when the condition is missing. For example, this code

```
if aeserflg=1 then do;  
  aesertxt='SERIOUS';  
end;  
else aeserflg=0 then do;  
  aesertxt='NON-SERIOUS';  
end;  
else do;  
  aesertxt='NOT SPECIFIED';  
end;
```

is replaceable with this

```
aesertxt=ifc(aeserflg, 'SERIOUS', 'NON-SERIOUS', 'NOT SEPCIFIED');
```

If *aeserflg* is 1 the condition is true so *aesertxt* is set to 'SERIOUS'. If *aeserflg* is 0 *aesertxt* becomes 'NON-SERIOUS'. If *aeserflg* is missing *aesertxt* becomes 'NOT SPECIFIED'.

A more complex example is

```
aestatus=ifc(aeserflg, 'AE is SERIOUS ' || ifc(aegraden>=3, 'and Grade 3 or  
Higher', 'but Grade 2 or Less'), 'AE is NOT Serious', 'AE Serious Flag  
Missing');
```

Here, if *aeserflg* is 1, the text *astatus* is set to either 'AE is SERIOUS and Grade 3 or Higher' or 'AE is SERIOUS but Grade 2 or Less' depending on whether *aegraden* is greater than or equal to 3 (the nested true condition) or less than 3 (the nested false condition). If *aeserflg* is 0, *astatus* is set to the outer IFC 'false' condition 'AE is NOT Serious', or if *aeserflg* is missing *astatus* is set to 'AE Serious Flag Missing'.

A 'sister' function to CHOOSEC is WHICHC, which returns the number of a matching text string. The first argument is the string to search for, and is followed by one or more arguments which are text strings to compare to.

```
conf_id=whichc(orgname, 'CDISC', 'CDASH', 'Global Forum', 'Pharmasug', 'Phuse');
```

If *orgname* is 'Pharmasug', *conf_id* is returned as 4, because 'Pharmasug' is the fourth item in the list. If more than one occurrence of the matching string is in the list, the number of the first occurrence is returned. If the string is not found in the list WHICHC returns 0.

There are also two corresponding numeric functions CHOOSEN and WHICHN, which select a numeric value from a list, and return the first numeric item in a list which matches the first argument value, respectively. The terms 'item' and 'value' can be a numeric constant, a missing value, or an arithmetic expression.

EXTENDING THE INDEX FUNCTION

Many SAS programmers are familiar with INDEX, a function which finds the location in a string (*str1*) of the first occurrence of a second string (*str2*) (*index(str1,str2)*). A limitation of INDEX is it only finds the first occurrence of the sub-string. Sometimes there is a need to find the location of the second or later occurrences, or refer from the end of the string instead of the beginning. For example, the SCAN('str1',n) function allows specification of a first, second, or later 'word' and, by using a negative number for *n*, the scan begins at the string end instead of the beginning. INDEX does not currently have such an ability. A work around this limitation is to use INDEX with SUBSTR, for example:

```
index(str1,substr(index(str1,str2)+1))
```

would return the position of the second occurrence of *str2* in *str1*, if there is one. Below is a macro, INDEXN, which evaluates the position of the *n*th occurrence of a second string in a first string. The first parameter *str1*, is the string to be searched, the second parameter, *str2*, is the sub-string being sought in *str1*, and the third parameter, *occ*, is the occurrence number (*n*) being sought, a negative value of *occ* performs the search from the end instead of the start of *str1*. *str2* must be shorter than *str1* and *occ* must be an integer and cannot be zero. The remaining two parameters, *fpos* and *ftext* contain the position from the start of the first character in *str2*, and the text in *str1* from that point. If the specified occurrence of *str2* is not found in *str1*, *fpos* is zero and *ftext* is blank. The code works by recursively calling the INDEX function on each nested SUBSTR. When *occ* is negative the REVERSE function is used to reverse the order of the contents of both strings. The TRIM function is needed to remove any trailing spaces so they are not placed as leading spaces in the reversed text. The INDEX function then performs the search from the start of the reversed string. To obtain the position of the first character of *str2* in *str1* the position less one and the length of *str2* less one are subtracted from the length of *str1*.

```
%macro indexn(str1=,str2=,occ=1,fpos=,ftext=);

  %local i;
  &fpos=0;
  %if &occ>0 %then %do;
    %do i=1 %to &occ; &fpos=&fpos+index(substr(&str1,&fpos+1),&str2); %end;
  %end;
  %else %if &occ<0 %then %do;
    %let occ=%eval(&occ*-1);
    %do i=1 %to &occ; &fpos=&fpos+
      index(substr(reverse(trim(&str1)),&fpos+1),reverse(trim(&str2)));
    %end;
    &fpos=length(&str1)-&fpos-length(&str2)+2;
  %end;
  &ftext=substr(&str1,&fpos);

%mend indexn;
```

For example, using this text string:

```
HEADING=Calling all SAS Programmers to Pharnasug 2019! Learning (& fun) for
all in Philadelphia PA on June 15/16-19 2019.
```

and this INDEXN call:

```
%indexn(str1=heading,str2='all',occ=3,fpos=x,ftext=subtext);
```

x (&*fpos*) would be 69, the location of the third occurrence of the sub-string 'all' (The first occurrence is in the word 'Calling'), and *subtext* (&*ftext*) would be the text from that point, i.e. 'all in Philadelphia PA on June 15/16-19 2019'.

There are two additional functions related to INDEX, INDEXW and INDEXC. INDEXW returns the position in a string of the first occurrence of a specified 'word' and INDEXC returns the first occurrence of any of a

specified sequence of characters, whether they are a whole word or within other characters. Examples are:

```
indexw(heading, 'Pharmasug')
```

returns 32, the position of the 'P' in 'Pharmsug' in *heading*.

INDEXC may have two or more strings to search for. For example

```
indexc('Pharmasug', 'arm') and indexc('Pharmasug', 'ram')
```

both return 3, the position of the first 'a' in 'Pharmasug', and

```
indexc('Pharmasug', 'sum', 'house')
```

returns 2, the first occurrence of any of the letters in 'sum' and 'house' is the 'h'

SORTING STRINGS

Sorting the observations in a data set based on the values of key variables is familiar and common to all SAS programmers, however, occasionally there is a need to sort items arranged horizontally in a data structure such as an array. The SORTC function sorts a string of character variables, or array elements, alphabetically. Example code is shown on the next page, it is a macro named WORDSORT which takes a single long string of words and sorts them. The macro is performed on a data set, *inds*, with *str1* being the input character string. The sorted string is returned in the string specified by the parameter *str2*. (This must not already be an existing variable in *inds*.) The first step is to determine the length of the input string and create a blank output string, *str2*, of that same length. The run-time current length of a variable (of any type) is determined with the VLENGTH function (see later). The SYMPUT function takes a run time value and stores it in a macro variable, which can be referenced in a later DATA step or PROCEDURE (not the current DATA step). Having defined *str2*, the next step is to set up an array of character type to store each word in each element. The array named WORDS is `_TEMPORARY_` because it does not have to map to actual SAS variables. Now the number of words, *n*, in *str1*, is counted, using the COUNTW function. If there are two or more words in *str1* they are loaded into the array WORDS starting at element 1 and ending at element *n*. Now CALL SORTC performs the sort of the array elements in alphabetical order. Finally, the sorted words are added sequentially to *str2*, each separated by a space (or other defined delimiter).

```

%macro wordsort(inds=, str1=, str2=sort, delim= );

%local maxw maxlen;
%let maxw=100;

data _null_;
  set &inds;
  call symput('maxlen',vlength(&str1));
run;

data &inds;
  set &inds;
  array words {&maxw} $100 _temporary_;
  length &str2 $&maxlen;
  do i=1 to &maxw;
    words{i}=' ';
  end;
  n=countw(&str1,"&delim");
  if 2<=n<=&maxw then do;
    do w=1 to n;
      words{w}=scan(&str1,w,"&delim");
    end;
    call sortc(of words{*});
    do i=dim(words)-n+1 to dim(words);
      if words{i} ne ' ' then do;
        &str2=trim(left(&str2))||"&delim"||trim(left(words(i)));
      end;
    end;
  end;
run;

%mend wordsort;

```

A few points to note here are the default word delimiter is a space, but the parameter *delim* may be used to define a different delimiter. The maximum permitted length of a character string in SAS is 32767 characters, in SAS version 6 and earlier the maximum length is only 200. Characters other than 'A'-'Z' and 'a'-'z' may be used but the exact sorting sequence depends on the character set in use (ASCII or EBDIC). The macro reinitializes the temporary array at the start of each DATA step iteration because a temporary array is RETAINED. The DIM function is used to determine the size of the array (should be the same as &maxw). An important issue when using SORTC on an array is the sort includes *all* (&maxw) of the elements, including unused (blank) elements. For this reason, when loading the sorted words into *str2* the sorted array is scanned from the first non-blank element, counted backwards from the length of the array, `dim(words)`)

Examples:

```

data ph_locs;
  length venues $200;
  venues='Denver Atlanta Portland Orlando Nashville Chicago Baltimore
        Seattle Philadelphia';

run;

%wordsort(inds=ph_locs,str=venues,str2=vensort);

```

The string *vensort* is created with the same length as *venues* and has the city names sorted alphabetically as:

```
`Atlanta Baltimore Chicago Denver Nashville Orlando Philadelphia Portland  
Seattle`
```

If there are any city names with two or more words a delimiter is required, for example;

```
data ph_locs;  
  length venues $200;  
  venues='Salt Lake City!Bonita Springs!Denver!Atlanta!Portland OR!  
  Orlando!Nashville!San Francisco!Chicago!San Diego!Baltimore!  
  Seattle!Philadelphia'  
  
run;
```

```
%wordsort(inds=ph_locs,str=venues,str2=vensort2,delim=!);
```

The string *vensort2* is created with the same length as *venues* and has the city names sorted alphabetically as:

```
`Atlanta!Baltimore!Bonita Springs!Chicago!Denver!Nashville!Orlando!  
Philadelphia!Portland OR!Salt Lake City!San Diego!San Francisco!Seattle`
```

Here the exclamation point (!) is the delimiter, hence *delim* is specified as '!' (Quotation marks are not required). After sorting, the delimiter can be removed by using the TRANSLATE function (see later). The following statement replaces spaces with underscores and then replaces the exclamation points with spaces.

```
vensort2=translate(translate(vensort2,'_',' '),',' ','!');
```

results in

```
`Atlanta Baltimore Bonita_Springs Chicago Denver Nashville Orlando  
Philadelphia Portland_OR Salt_Lake_City San_Diego San_Francisco Seattle`
```

REMOVING EXCESS SPACES

Sometimes there is a need to consolidate multiple spaces into one or to remove excess blanks. These are the functions to use:

COMPRESS(*<string>*, *<character list>*) removes all of the occurrences of each character in *character list* from *string*. The default *character list* is the space. Examples are

```
compress(`Pharma      sug 2019`);
```

results in 'Pharmasug2019'

```
compress(`CYCLE 1XX  DAY 15A`,`AX`);
```

results in 'CYCLE 1 DY 15', (spaces are not compressed).

A character string may be left or right justified using the LEFT and RIGHT functions.

LEFT(*string*) removes all leading spaces (left justifies)

RIGHT(*string*) right justifies to the declared string length, for example if the string *str* has a declared length of \$25 and a value of 'Pharmasug 2019', **RIGHT**(*str*) will result in ' Pharmasug 2019', with 11 leading spaces because the length of *str* is \$25 and length(*str*) (when left justified) is 14.

COMPBL removes leading and trailing blanks and replaces multiple blanks with a single space. This function also has a second optional argument, which is a list of one or more characters to remove any multiple occurrences of and replace with a single occurrence. Examples are:

```
compbl('  Pharmasug      2019');
```

results in 'Pharmasug 2019'

```
compbl('*** Mind Puzzler ***', 'z');
```

results '*** Mind Puzler ***'

```
compbl('*** Mind Puzzler ***', 'z*');
```

results '* Mind Puzler *'

When a second argument is specified spaces are no longer processed.

STRIP removes all leading and trailing blanks. TRIM removes all trailing blanks. A point to note is that whenever a SAS character string is stored in a character variable which is longer than the string length it is always padded with trailing spaces. When concatenating two character texts the TRIM (or the STRIP) function should always be used on the first string text during the concatenation, for example if *parta* and *partb* are two text strings being joined to form one string *partab*:

```
partab=trim(parta)||' '|partb;
```

Is not the same as

```
parta=trim(parta);
```

```
partab=parta||' '|partb;
```

In this case any trailing blanks are stored in *parta* after TRIM(*parta*) and will result in the trailing blanks being inserted between *parta* and *partb*.

A difference between the STRIP and TRIM functions is when the argument has no non-blank characters STRIP returns a null string, of length zero, whereas TRIM returns a single space.

JOINING TEXTS AND CONCATENATION

A useful set of string concatenating functions are CAT, CATS, CATT, and CATX

CAT joins text strings without removing any leading or trailing spaces.

CATT joins text strings after removing the trailing blanks

CATS joins text strings after stripping both the leading and trailing blanks

CATX joins text strings after stripping both the leading and trailing blanks and inserts a specified delimiter

For example, if there are three non-blank character strings *str1*, *str2*, and *str3* to be concatenated

cat(of *str1-str3*) is the equivalent of *str1||str2||str3*

catt(of *str1-str3*) is the equivalent of trim(*str1*)||trim(*str2*)||trim(*str3*)

cats(of *str1-str3*) is the equivalent of strip(*str1*)||strip(*str2*)||strip(*str3*)

catx(' ', of *str1-str3*) is the equivalent of strip(*str1*)||' '||strip(*str2*)||' '||strip(*str3*)

In the last example, CATX, the first argument is the delimiter, in this case, a space.

A fifth member of the CAT functions is CATQ, which concatenates and adds quotation marks to each string containing a specified delimiter.

More information about the CAT functions can be found in the SAS documentation or as indicated under 'References and Suggested Reading'.

Lastly, the REPEAT function may be used to repeat a specified character or sequence of characters, for example:

```
banner=repeat('*',100);
```

stores a line of 100 '*'s in the string *banner* (which must have a length of at least \$100), and

```
banner=repeat('<-+>',5);
```

causes *banner* to be '<-+> <-+> <-+> <-+> <-+>';

CHANGING THE CONTENTS OF A STRING (TRANSLATING)

The TRANSLATE function described earlier is used to replace one character with another. Its general format is

```
translate(<string>,<repl>,<ch1>,<rep2>,<ch2>,...<repn>,<chn>);
```

This function replaces every occurrence of *ch1* with *rep1*, and, if specified, *ch2* with *rep2* and so on. Literal characters must be enclosed in quotation marks, and may be spaces or punctuation characters. TRANSLATE only replaces single characters, a related function, TRANWRD, replaces each occurrence of a specified 'word', delimited by spaces, with a replacement text string. The syntax is

```
tranwrtd(<string>,<old word>,<new word>);
```

The old and new texts may be of different lengths and either may be a single character or blank. Three important points to note are (1) the text being replaced is specified before the replacement text, the opposite is true for TRANSLATE, and (2) unlike TRANSLATE, only one replacement can be made with each function call – there are always three arguments, and (3) if the replacement txt is longer than the text being replaced the string holding the result must be of sufficient length otherwise truncation will occur.

For example, using the *heading* text:

```
HEADING='Calling all SAS Programmers to Pharmasug 2019! Learning (& fun) for all in Philadelphia PA on June 15/16-19 2019.'
```

The function call

```
heading=tranwrtd(heading,'call','bigtime');
```

results in *heading* becoming

```
'Calling bigtime SAS Programmers to Pharmasug 2019! Learning (& fun) for bigtime in Philadelphia PA on June 15/16-19 2019.'
```

Another 'sister' translation function is TRANSTRN. This has the same syntax and modus operandi as TRANWRD except that any occurrences of the old string, including within other text, are replaced. If TRANSTRN instead of TRANWRD had been used in the above example the word 'Calling' would have been changed to 'Cbigtimeing' in addition to the words 'all' being changed to 'bigtime'.

MISCELLANOUS EXTRAS

The functions RANK and BYTE link a character to its collating sequence number (used by SORTC). RANK(*cvar*) returns the ASCII or EBDIC ranking number for the character *cvar*, and BYTE(*n*) returns the character with the collating sequence number *n*. Examples are:

```
letter=byte(65);
```

returns *letter* equal to 'A'

```
n=rank('B');
```

returns *n* equal to 66.

DATA STEP EXTRAS

DETERMINING ATTRIBUTES FROM WITHIN A DATA STEP

The functions described under this heading are for determining the attributes of a column during DATA step or SAS PROCedure execution. In this way code and conditional execution can be tailored to an attribute, such as the declared length of a string. In the WORDSORT macro described above, VLENGTH is used to determine the declared length of a variable (as opposed to the LENGTH function, which returns the length of the current string contents less trailing spaces). There is no need in these cases to use SASHELP or DICTIONARY.COLUMNS, which has to be performed outside of the DATA step. These 'V' functions directly access the SASHELP contents at run time. The attribute value returned may have been declared in the current or a prior DATA step.

In addition to VLENGTH there are the following:

VLABEL(<varname>) returns the variable's label (blank if none specified)

VTYPE(<varname>) returns the single character 'C' if *varname* is character, or 'N' if *varname* is numeric

VFORMAT(<varname>) returns the format of the variable, including user defined formats in PROC FORMAT, as a text string (including the '.') e.g. '\$100.', '8.3', 'DATE9.', 'IS8601DT.'

VVALUE(<varname>) returns the formatted value corresponding to the variable's actual value. For example, if the numeric variable *evntdt* has an actual value of 20000 and a format of DATE9.

VVALUE(*evntdt*) would return the text string '04OCT2014'.

VNAME(<array reference>) returns the variable name corresponding to an array element. For example

```
array vitals {6} pulse resp sbp dbp temp weight;
vname(vitals{3})
```

would return 'sbp'.

DETERMINING SAS OPTIONS FROM WITHIN A DATA STEP

The GETOPTION facility allows a setting from the most recent prior OPTIONS statement to be read at run time. The following example lists the PAGESIZE, LINESIZE, CENTER/NOCENTER, and MPRINT/NOMPRINT settings.

```
paglines=getoption('pagesize');
linecols=getoption('linesize');
centered=getoption('center');
mtrace=getoption('mprint');
put paglines= linecols= centered= mtrace=;
```

```
PAGELINES=60  LINECOLS=132  CENTERED=CENTER  MTRACE=NOMPRINT
```


CARRYING FORWARD FROM PRIOR OBSERVATIONS

Quite often in the SAS DATA step there is a need to refer back to the values of a column in a prior observation. An example is calculating the change since a prior reading. Two functions to do this are LAGn and DIFn. LAGn retrieves the *n*th prior value. The default value of *n* is 1, the immediately prior observation. LAGn differs from RETAIN in that each value of the column read into the Program Data Vector (PDV) is placed in a queue and kept for reference at the next *n*th observation. A value assigned earlier in the current DATA step is not added to the queue, so the LAGn value will be missing or the prior value in the input dataset in that case. DIFn is very similar to LAGn but this function keeps the difference between the current and *n*th prior values. For both of these functions *n* can be in the range 1 to 100. If *n* refers to before the first observation (*_n_*=1), for example LAG3 is used on the second (*_n_*=2) observation, a missing value is returned.

An example is shown below using weight data collected at weekly intervals for a given patient in a study. The PTWT DATA step is just to set up the example, in practice there could be many weeks of a study for many patients.

```
data ptwt; /* Set up example data */
    patnum=1000;
    study_wk=1;   wght_kg=82.4;   output;
    study_wk=2;   wght_kg=83.8;   output;
    study_wk=3;   wght_kg=83.1;   output;
    study_wk=4;   wght_kg=80.7;   output;
    study_wk=5;   wght_kg=81.4;   output;
run;
```

The data must now be sorted in patient and study week order to evaluate the change in weight over the course of the study.

```
proc sort data=ptwt;
    by patnum study_wk;
run;
```

The DATA step shown overleaf uses the LAG function to show the patient's weight for the prior week and for two weeks prior, and uses the DIF function to show the corresponding changes in weight. In the first observation for each patient there is no prior weight and care must be taken not to roll over the last weight belonging to the prior patient. The numeric variable *count* keeps track of the current observation number for the current patient (outer BY group) and the LAG and DIFF functions are only called for the second and subsequent observations. LAG2 and DIFF2 are used to measure the weight change from two weeks prior, so these values are calculated only on and after the third observation for the patient. The RETAIN statement is used to retain the first (baseline) weight, from the first observation for the patient. Subtracting this from the current weight gives the change from baseline. The *n* in LAGn and DIFFn can only be a numeric constant, it cannot change dynamically as the DATA step observations are read.

```

data ptwtchg(drop=basewght count);
  set ptwt;
  by patnum study_wk;
  retain basewght count 0;
  if first.patnum then do;
    basewght=wght_kg;
    count=0;
  end;
  count=count+1;
  if count ge 2 then do;
    priorwt=lag(wght_kg);
    chgpriwt=dif(wght_kg);
  end;
  if count ge 3 then do;
    priorwt2=lag2(wght_kg);
    chgpriw2=dif2(wght_kg);
  end;
  chgbl=wght_kg-basewght;
run;

```

The resulting output for PATNUM 1000 is:

PATNUM	STUDY_WK	WGHT_KG	PRIORWT	PRIORWT2	CHGPRIWT	CHGPRIW2	CHGBL
1000	1	82.4	0.0
1000	2	83.8	82.4	.	1.4	.	1.4
1000	3	83.1	83.8	82.4	-0.7	0.7	0.7
1000	4	80.7	83.1	83.8	-2.4	-3.1	-1.7
1000	5	81.4	80.7	83.1	0.7	-1.7	-1.0

A final point to note is there is no corresponding LEAD or DIF lead function to read the next or later observations because the data in an observation is only known to the system when it is in the PDV.

There are 'work a rounds' to simulate a LEAD function, the most obvious of which is to sort the data in reverse order, in this case the BY statement would read

```
by patnum descending study_wk;
```

and then use LAGn and DIFFn on the reverse ordered observations before re-sorting the data back into ascending sequence. This, and other methods are discussed in several other SAS Conference Proceedings, some of which are referenced in the 'References and Suggested Reading' section.

DATE AND TIME EXTRAS

Here are a few useful date and time functions.

DAY returns the day number of the month in a SAS internal numeric date (or DATEPART of a SAS numeric datetime), as a numeric value from 1 to 31.

WEEKDAY returns the day of the week for a given internal numeric SAS date. The numeric result returned is in the range 1 through 7, where 1 is Sunday and 7 is Saturday. The following code obtains the weekday name for the current date by using the CHOOSEC and WEEKDAY functions:

```
today_is= choosec(weekday(today()), 'Sunday', 'Monday', 'Tuesday', 'Wednesday',  
    'Thursday', 'Friday', 'Saturday');
```

Similarly, the MONTH function returns the month number, 1 through 12, from a SAS internal numeric date. This code returns the month name in the SAS internal date *evntdt*:

```
Month_of= choosec(month(evntdt), 'January', 'February', 'March', 'April',  
    'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December');
```

QTR returns the quarter, and YEAR the four-digit year. For example, this code writes the last day of the quarter and the four-digit year from the SAS numeric date *buddaten* to the character string *closdate*:

```
closdate=trim(choosec(qtr(buddaten), 'March 31', 'June 30',  
    'September 30', 'December 31')) || ' ' || put(year(buddaten), 8.);
```

The HOLIDAY function returns the calendar date as a numeric SAS internal date for a specified holiday, for example, this code returns '25DEC2019' to the character string *holidate*:

```
holidate=put(holiday('christmas', 2019), date9.);
```

The year needs to be specified because some holidays are tied to a specific weekday in a month (e.g. Labor Day is the first Monday in September) and other holidays are determined because of religious observances (e.g. Good Friday and Easter Sunday). In these situations, the date varies from one year to the next. A point to note here is that 'observed' holidays are not evaluated, for example, if New Year's Day is on a Sunday the date returned is still January 1, not the following Monday January 2. A detailed list of all the holiday names (in the USA and other countries) which can be used with the HOLIDAY function is available under support.sas.com/documentation.

HOUR returns a number from 0 to 23, the hour in a SAS internal numeric time, or the timepart of a datetime. MIN returns the minute, a number from 0 to 59.

HMS(*hr,min,sec*) returns a SAS numeric internal time from the arguments *hr*, *min*, and *sec*. *hr* must be a number 0 through 23, *min* a number 0 through 59, and *sec* a number 0 through 59.

CONCLUSION

This paper has introduced some methods and techniques which SAS programmers should find useful to improve code quality and efficiency and to reduce programming effort. Users are encouraged to share this paper's contents with co-workers, classmates, and instructors.

REFERENCES AND SUGGESTED READING

Online SAS Support <http://support.sas.com/documentation/cdl/en/lrdict/>

Joshua M. Horstman, Nested Loop Consulting, Indianapolis, IN, PharmaSUG 2017 - Paper BB21 "Beyond IF THEN ELSE: Techniques for Conditional Execution of SAS®Code".

<https://www.lexjansen.com/pharmasug/2017/BB/PharmaSUG-2017-BB21.pdf>

Mark Keintz, Wharton, Research Data Services, SAS Global Forum Proceedings 2017 - Paper 1277-2017 "Leads and Lags: Static and Dynamic Queues in the SAS® DATA STEP" 2nd edition.

<http://support.sas.com/resources/papers/proceedings17/1277-2017.pdf>

CodeJoshua M. Horstman, Nested Loop Consulting, Indianapolis, IN, Midwest SAS Users Group MWSUG conference 2017- Paper BB071 "Fifteen Functions to Supercharge Your SAS® Statements".

<https://www.pharmasug.org/proceedings/2018/BB/PharmaSUG-2018-BB17.pdf>

Thomas E. Billings, Union Bank, San Francisco, California, Pharmasug 2012 – "IFC and IFN Functions: Alternatives to Simple DATA Step IF-THEN-ELSE, SELECT-END Code and PROC SQL CASE".

<https://www.lexjansen.com/wuss/2012/28.pdf>

Ronald Cody, Ed.D, Northeast SAS Users Group (NEWSUG) 2009 – "An Introduction to SAS® Character Functions."

<https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/217-2007.pdf>

Louise S. Hadden, Abt Associates Inc., Northeast SAS Users Group (NEWSUG) 2009 - "Purrfectly Fabulous Feline Functions".

<http://support.sas.com/resources/papers/proceedings10/205-2010.pdf>

Neil Howard, Independent Consultant, Charlottesville, VA, SAS Global Forum Proceedings 1999 (SUGI 24) – Paper 57 - "Introduction to SAS(r) Functions"

<https://support.sas.com/resources/papers/proceedings/proceedings/sugi24/Begtutor/p57-24.pdf>

Timothy J Harrington, DataCeutics, Inc., PharmaSUG 2018 - Paper EP-16 "Discover the Deeper DATA Step" <https://www.pharmasug.org/proceedings/2018/EP/PharmaSUG-2018-EP16.pdf>

ACKNOWLEDGMENTS

The author would like to express gratitude to the following for their assistance in reviewing this paper: Jeffrey Dickinson, Sharon Hall, and Kathy Greer at Navitas, Inc. 1610 Medical Drive, Suite 300, Pottstown, PA 19464 | 610.970.2333 | info@dataceutics.com. Richann Watson and Josh Horstman, PharmaSUG 2019 Academic Chair, at AcademicChair@PharmaSUG.org

CONTACT INFORMATION

Your comments and questions are valued and encouraged. You may contact the author at:

Timothy J. Harrington

Navitas, Inc.

1610 Medical Drive, Suite 300

Pottstown, PA 19464

Office: 610 970 2333

harringt@dataceutics.com

www.dataceutics.com

DataCeutics was acquired by Navitas Life Sciences in March of 2019

Any brand and product names are trademarks of their respective companies.

