Paper PP23

SAS® to R: CHEAT SHEET – A Practical Guide for Clinical Programmers

Lekitha JK, Atorus Research, Bangalore, India
Jalendhar Ettedi, Atorus Research, Bangalore, India

ABSTRACT

As the use of R in clinical research continues to rise particularly for tasks such as data visualization, statistical modelling, and automation clinical programmers who have traditionally been trained in SAS are finding themselves needing to operate in environments where both programming languages are utilized. This paper introduces a cheat sheet that transitions SAS techniques to R, serving as a quick-reference tool that aligns common SAS procedures, data step logic, and reporting methods with their corresponding functionalities in R. The cheat sheet highlights essential clinical programming activities, including data import and export, manipulation of datasets, derivation of variables, merging, flagging, and the creation of TLFs. By providing side-by-side code comparisons, we illustrate popular SAS procedures like PROC SORT, PROC TRANSPOSE, and DATA STEP logic alongside their R counterparts using packages such as dplyr, tidyr, etc.

INTRODUCTION

R is a language and environment for statistical computing and graphics. But why should you want to use it? First and foremost, R is open-source, and available on every major platform. Because of this, when you do your analysis in R, anyone can easily replicate it.

In clinical programming when code is treated as proprietary, multiple people are solving the same problem across the industry. For those of you in the pharma world, think about how CDISC gave us data standards. This provided us a unified approach everyone could embrace, eliminating the need for companies to spend time deciding how to structure the data, and dramatically lowering review times for regulatory agencies.

What if we could standardize code too? If open-source code were introduced into the industry, we could stop focusing on the redundant things every organization needs to create their own approach for and focus on science. Users can share solutions and benefit from the lessons learned that other programmers have already overcome.

It's not difficult to find resources to help you understand how to program in R. But there is so much that R can do that those resources can be overwhelming. This paper will help you transition your programming knowledge in SAS® to programming knowledge in R.

This paper answers the basic questions of "How does R work?" You'll see throughout this paper that your base knowledge of SAS® will have you programming in R in no time.

INSTALLING PACKAGES AND LOADING LIBRARIES

You can install a package using the `install.packages()` function. The input to the function is the name of the package you want to install inside quotations.

This example installs a package named `pharmaRTF`.

```
install.packages("pharmaRTF")
```

There are several ways that package installation and management can be handled, most of which is outside the scope of this paper. Without changing any defaults, packages only need to be installed once for your user account. Updates to packages need to be re-installed as necessary. Within a production environment, there should be strict governance around how package management is handled, and the installation of packages is something that you may not have control over.

Once you have the package installed, you can use its functionalities.
When you only need occasional use of a few functions, you can access them with the notation `packagename::functionname()`.

This example uses the `write_rtf()` function from the `pharmaRTF` package.

```
pharmaRTF::write_rtf(<inputs to function>)
```

Packages can occasionally have functions by the same name. Using the `packagename::functionname()` notation explicitly defines which package the function comes from. So using this notation not only makes your code clearer in these scenarios but also allows you to work around potential function name conflicts – referred to as namespace conflicts.

If you're going to make more intensive use of the package, you can import the package into your R session.

You can load a library using the `library ()` function and specify the package name as the input to the function. Note that quotations are not required in the `library ()` function. When you load a library, any of the functions in that package will be available for you to use by simply calling the function.

This example loads the `pharmaRTF` package, and then uses the `write_rtf()` function.

```
library(pharmaRTF)
write_rtf(<inputs to function>)
```

Note, while you only need to install a package once, you need to load a library every time you start a new R session.

READING IN FILES

READING IN XPTS AND SAS7BDATS

In R, data can be read using two functions from the `haven` package. The `read_xpt()` function reads in XPT files and the `read_sas()` function reads in sas7bdat files.

Both functions require you to specify the path of the data, similar to a SAS® libname statement.

In this example we are creating a dataframe called dm from the demo XPT file. For the path we would specify the directory path of demo.xpt.

```
dm <- read_xpt("<path>/demo.xpt")
```

And similarly, in this example we are creating a dataframe called dm from the demo sas7bdat file. For the path we would specify the directory path of demo.sas7bdat.

```
dm <- read_sas("<path>/demo.sas7bdat")
```

This example would look for demo.sas7bdat within your current working directory.

```
dm <- read_sas("./demo.sas7bdat")
```

Just as the tools to read in data using SAS® have various options, the `read_xpt()` and the `read_sas()` functions have several optional parameters to control how you can read in your data.

READING IN AN XLS/XLSX FILE

The `read_excel()` function from the `readxl` package reads in xls and xlsx files. In it's most basic use you simply specify the path to the Excel file.
Consider this xlsx file containing Adverse Events of special interest. The file has 2 sheets, AESI1 and AESI2.

	A	B
1	TERM	CRITERIA1
2	DIARRHOEA	
3	DYSPEPSIA	Y
4	DYSPHONIA	
5	DYSPNOEA	Y
6	EPISTAXIS	Y
7	ERYTHEMA	

AESI1 AESI2

	A	B
1	TERM	CRITERIA2
2	DIARRHOEA	
3	DYSPEPSIA	
4	DYSPHONIA	Y
5	DYSPNOEA	
6	EPISTAXIS	Y
7	ERYTHEMA	Y

AESI1 AESI2

In this example we are creating a dataframe called aesi1 from the aesi xlsx file. For the path we would specify the directory path of aesi.xlsx.

```
aesi1 <- read_excel("./data/aesi.xlsx")
```

Notice in this example we did not specify which sheet to read in. As we saw, in its most basic use you only need to specify the path to the Excel file. By default, the first sheet is read in.

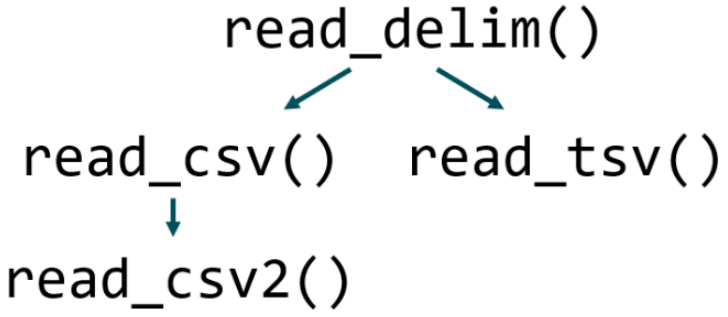
You can specify which sheet to import by using the `sheet` parameter. Here we are creating a dataframe called aesi2 from the aesi xlsx file, using the AESI2 sheet.

```
aesi2 <- read_excel("./data/aesi.xlsx", sheet = "AESI2")
```

In addition to which sheet to read in, the `read_excel()` function has parameters that allow you to specify many other things such as the column names, the column types, which rows to read in and more - giving you the flexibility to read in your xls or xlsx files as needed.

READING IN A DELIMITED FILE

Delimiter-separated files contain data where the values are separated with a specific delimiter. A delimiter is a character that specifies how columns should be separated inside the data file.



The `read_delim()` function from the `readr` package reads in a delimited file.

The `read_csv()` and `read_tsv()` functions are special cases of the `read_delim()` function. They read in the most common types of flat file, comma separated values, abbreviated csv, and tab separated values, abbreviated tsv.

In this first example we are creating a dataframe called aesi1 from the aesi1 csv file. For the path we would specify the directory path of aesi1.csv.

```
aesi1 <- read_csv("./data/aesi1.csv")
```

Similarly in this second example we are creating a dataframe called aesi2 from the aesi2 csv file. For the path we would specify the directory path of aesi2.csv.

```
aesi2 <- read_csv("./data/aesi2.csv")
```

These examples illustrate using the `read_csv()` function with minimal inputs, now we'll see some examples where a few of the optional parameters are used.

We've now seen how to read in a csv file using the `read_csv()` function. That logic can be easily translated to the `read_tsv()` function, except we'll need to read in a tab delimited file instead of a comma delimited file.

FINALIZING AND OUTPUTTING FILES

APPLYING METADATA

Often when programming you will need to apply metadata, such has dataframe labels, column labels, and column lengths. The R dataframe typically carries less metadata than a SAS® dataset. But in clinical programming we need to use metadata, so luckily there are several functions you can use to apply any of the metadata you need.

Consider this dataframe.

adsl:

USUBJID	AGE	TRT01P	TRT01PN	TRTSDT	TRTEDT
01-351-515	64	Placebo	0	2019-07-06	2020-01-03
01-351-528	69	Miracle High Dose	9	2019-01-16	2019-07-14
01-351-533	77	Miracle Low Dose	6	2019-09-16	2019-09-29
01-351-534	80	Miracle High Dose	9	2020-01-01	2020-07-01
01-351-597	71	Miracle Low Dose	6	2019-07-03	2020-01-08

Example: We can assign a dataframe label using the `xportr_df_label()` function from the `xportr` package. The function assigns a dataframe label from a specified dataset level metadata dataframe.

```
df_labels <- tribble(
  ~dataset, ~label,
  "adsl",   "Subject-Level Analysis Dataset"
)

adsl <- adsl %>%
  xportr_df_label(df_labels)
```

OUTPUTTING DATASETS

The `xportr_write()` function from the `xportr` package writes a local dataframe into SAS® transport file of version 5. In its most basic use you simply specify the dataframe name, and the path where you want to write the XPT file.

Suppose we have this adsl dataframe. We can output the dataframe and its metadata in to an XPT file.

adsl:

USUBJID	AGE	TRT01P	TRT01PN	TRTSDT	TRTEDT
01-351-515	64	Placebo	0	2019-07-06	2020-01-03
01-351-528	69	Miracle High Dose	9	2019-01-16	2019-07-14
01-351-533	77	Miracle Low Dose	6	2019-09-16	2019-09-29
01-351-534	80	Miracle High Dose	9	2020-01-01	2020-07-01
01-351-597	71	Miracle Low Dose	6	2019-07-03	2020-01-08

In this example we are creating a file called adsl.xpt from the adsl dataframe. For the path we specify the path where we want to output the XPT.

```
xportr_write(adsl, "./output/adsl.xpt")
```

OUTPUTTING AN XLSX FILE

The `write.xlsx()` function from the `openxlsx` package writes a dataframe or a list of dataframes to an xlsx file. In it's most basic use you simply specify the dataframe name, and the path where you want to write the xlsx file.

Consider this subjects dataframe.

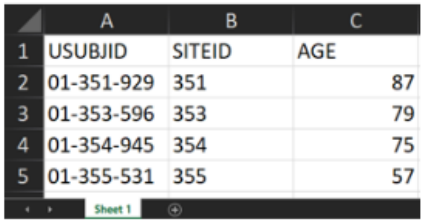
subjects:

USUBJID	SITEID	AGE
01-351-929	351	87
01-353-596	353	79
01-354-945	354	75
01-355-531	355	57

In this example we are creating a file called subjects.xlsx from the subjects dataframe. For the path we specify the path where we want to output the xlsx.

```
write.xlsx(subjects, "./output/subjects.xlsx")
```

In our xlsx file we have one sheet named “Sheet 1” containing the contents of the subjects dataframe.



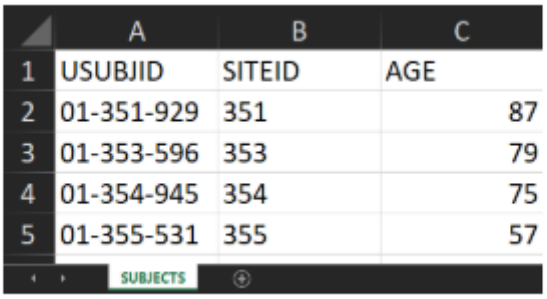
	A	B	C
1	USUBJID	SITEID	AGE
2	01-351-929	351	87
3	01-353-596	353	79
4	01-354-945	354	75
5	01-355-531	355	57

Sheet 1

You can specify a sheet name using the `sheetName` parameter.

Here we are creating a file called subjects2.xlsx from the subjects dataframe and naming the sheet “SUBJECTS.”

```
write.xlsx(subjects, "./output/subjects2.xlsx", sheetName = "SUBJECTS")
```



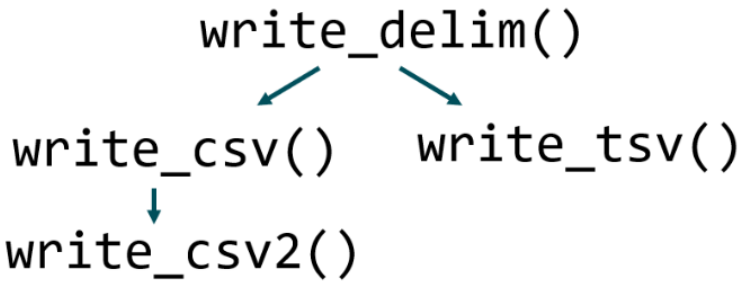
	A	B	C
1	USUBJID	SITEID	AGE
2	01-351-929	351	87
3	01-353-596	353	79
4	01-354-945	354	75
5	01-355-531	355	57

SUBJECTS

These were just a few examples to demonstrate the basic use of the `write.xlsx()` function.
The `openxlsx` package gives you a great deal of functionality to customize xlsx files as desired. With the package you can modify cells directly, introduce advanced formatting functionality, add Excel functions, and even do things like conditional formatting.

OUTPUTTING A DELIMITED FILE

The delimiter-separated files contain data where the values are separated with a specific delimiter. Just as there were several functions to read in various delimiter-separated files, there are several related functions that will write them.



The `write_delim()` function from the `readr` package writes a delimited file.

The `write_csv()` and `write_tsv()` functions are special cases of the `write_delim()` function. They write the most common types of flat file, csv files, and tsv files. csv is the abbreviation for comma separated values and tsv is the abbreviation for tab separated values.

Let's take a closer look at the `write_csv()` and `write_tsv()` functions.

Let's continue with the subjects dataframe.

subjects:

USUBJID	SITEID	AGE
01-351-929	351	87
01-353-596	353	79
01-354-945	354	75
01-355-531	355	57

In this example we are creating a file called subjects.csv from the subjects dataframe. For the path we specify the path where we want to output the csv.

```
write_csv(subjects, "./output/subjects.csv")
```

In this example we are creating a file called subjects.txt from the subjects dataframe. For the path we specify the path where we want to output the txt.

```
write_tsv(subjects, "./output/subjects.txt", col_names = FALSE)
```

The functions introduced in this section are very similar. Although we demonstrated examples with the `write_csv()` and `write_tsv()` functions, the logic would translate to any of the other functions.

TIDY SELECTION

Tidy selection, often referred to as "select helpers," is a tool that makes it easy to work with the columns of a dataframe. It provides a dialect of R for selecting columns based on their names or properties. Understanding tidy selection will help you write code in an understandable and concise way.

The following selection helpers from the tidyr package select columns by matching patterns in their names:

- `starts_with()`: selects columns that start with a prefix.
- `ends_with()`: selects columns that end with a suffix.
- `contains()`: selects columns that contain a literal string.
- `matches()`: selects columns that match a regular expression.
- And `num_range()`: selects columns that match a numerical range like x01, x02, x03.

Several operators are also utilized to simplify the process of selecting columns:

- A colon `(:)` selects a range of consecutive columns.
- An exclamation point `(!)` takes the complement of a set of columns.

- Ampersand and pipe or vertical bar `(& and |)` selects the intersection or the union of two sets of columns.
- And the `c()` function combines selections.

Tidy selection is used in various functions so you can easily select columns.Select helpers should be noted, and often the arguments will list `<tidy-select>` on the arguments in which the select helpers are valid.

MISSINGS

In R, there are two different ways of representing missing data.

The symbol `NaN` is a special type of missing for the double data type. It represents “not a number” and is the result of performing calculations that lead to undefined results. The `is.nan()` function can be used to indicate which elements are `NaN`.

The symbol `NA` is the general representation of missing data for all data types. It represents “not applicable” and is the result of data being missing for an unknown reason. In clinical programming it often means it simply wasn’t collected. The `is.na()` function can be used to indicate which elements are `NA`.

Let’s look more closely at NA.

Here we create a vector with 7 elements, two of which are missing values.

```
x <- c(1:3, NA, "five", NA, "seven")
x
```

```
## [1] "1"     "2"     "3"     NA      "five"  NA      "seven"
```

Using the `is.na()` function on our vector returns a logical vector which shows `TRUE` for the elements that are missing.

```
is.na(x)
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

As always with clinical programming it is important to understand the data you are using, so the ability to identify `NaN, NA,` and empty strings will ensure you handle them properly in programming.

FACTORS

Factors are essentially vectors which are specialized for categorical data. R stores the content of a factor as a vector of integers, for which an integer is assigned to each of the possible levels. Levels are the different categories contained in a factor. By default, R will organize the levels in a factor in alphabetical order, but factor levels can also be specified in whatever order you desire.

Consider this dataframe containing character columns for USUBJID and RACE.

```
## # A tibble: 2 x 2
##   USUBJID    RACE
##   <chr>      <chr>
## 1 01-351-703 BLACK OR AFRICAN AMERICAN
## 2 01-351-711 WHITE
```

We can convert the RACE column to a factor using the `factor()` function. The `factor()` function is used to encode a vector as a factor. Here we specify ordered levels of BLACK OR AFRICAN AMERICAN, ASIAN, and MULTIPLE.

```
races$RACE <- factor(races$RACE, levels=c("BLACK OR AFRICAN AMERICAN","ASIAN","MULTIPLE"))
```

Notice some syntax we haven't used before. The dollar sign `($)` allows you to extract a column from a dataframe as a vector. So this is saying to take the RACE column from the races dataframe.

Notice the levels parameter. We're using the `c()` function to specify multiple levels. We're also specifying levels for ASIAN and MULTIPLE which are not in our data. Remember, factors are used for categorical data, so we often know the full list of expected values. Including all values ensures if those values are ever added to the data, we'll have levels ready for them. A potential issue, however, is that we have a value of WHITE in our races dataframe, but we didn't give it a level.

If we look at the updated races dataframe, USUBJID is still a character vector, but RACE is now a factor.

```
## # A tibble: 2 x 2
##   USUBJID    RACE
##   <chr>      <fct>
## 1 01-351-703 BLACK OR AFRICAN AMERICAN
## 2 01-351-711 <NA>
```

Notice where we had a value of WHITE before we now have a missing value represented by the symbol `NA`. This is because we didn't specify a level for WHITE. Any values that do not have a level will be silently converted to `NA`.

And finally, if you want to check the levels, you can use the `levels()` function.

```
levels(races$RACE)
```

```
## [1] "BLACK OR AFRICAN AMERICAN" "ASIAN"
## [3] "MULTIPLE"
```

This was just an introduction to what factors are and how to create them.

OPERATORS

An operator is a symbol that tells the compiler to perform specific operations. In R there are several types of operators. Let's start with some of the most used operators, the assignment operators.

Assignment operators assign a value to a name. There are two types:

- The arrow operator `(<-)`

- And the equal operator `(=)`

Between the two types, arrow operators are preferred and recommended for use in R programming style guides for readability purposes since they clearly state which side you are making the assignment.

Another commonly used operator is the pipe operator `(%>%)`. The pipe operator from the `magrittr` package allows you to chain multiple calls into a single statement, without needing objects to store the intermediate results. The operator automatically takes the object returned from the left side of the pipe and applies it as the first argument on the right side of the pipe. It acts as a connector, like the phrase “and then” and allows you to read the code as a series of actions. For example, “first do x and then do y and then do z.”

Now that you’ve been introduced to the assignment and pipe operators, let’s look at a few examples of how they’re used.

In the first example we use the arrow operator to assign the text string “text” to an object named obj1.

```
obj1 <- "text"
```

In the second example we use the arrow operator to assign the numeric values of 1 and 2 to an object named obj2.

```
obj2 <- c(1,2)
```

Note, the `c()` function simply combines its arguments into vector form.

And finally, in the third example we use the arrow operator to assign the scalar object obj1 to an object named obj3.

```
obj3 <- obj1
```

ARITHMETIC OPERATORS

Arithmetic operators act on each element in a vector.

This table contains the arithmetic operators and their descriptions.

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation
x %% y	modulus (x mod y) 7%%2 is 1
x %/% y	integer division 7%/%2 is 3

In this example we are creating a new dataframe, labs3, from our existing dataframe, labs, and adding a new column named ADD which contains the values of AVAL plus 10.

Labs:

AVAL	BASE
142.0	140.0
4.4	4.5
103.0	106.0

```
labs3 <- labs %>%
  mutate(ADD = AVAL + 10)
```

AVAL	BASE	ADD
142.0	140.0	152.0
4.4	4.5	14.4
103.0	106.0	113.0

The addition operator `(+)` adds 10 to every value in the AVAL vector.The assignment operator assigns the results to an object named labs3.And the pipe operator allows us to read this as “take labs and then mutate it.” Since the pipe automatically takes the object returned from the left side of the pipe and applies it as the first argument on the right side of the pipe, we do not need to specify the dataframe as the first argument of the `mutate()` function.

LOGICAL OPERATORS

Like arithmetic operators, logical operators also act on each element in a vector. They return values that are of the logical datatype.

This table contains the logical operators and their descriptions.

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!	not
x | y	x OR y
x & y	x AND y

We’ll continue using the labs dataframe which contains columns for AVAL and BASE.

In this example we are creating a new dataframe, labs5, from our existing dataframe, labs, and adding a new column named LOGIC which contains the logical evaluation of whether each value of AVAL is greater than 100.

```
labs5 <- labs %>%
  mutate(LOGIC = AVAL > 100)
```

AVAL	BASE	LOGIC
142.0	140.0	TRUE
4.4	4.5	FALSE
103.0	106.0	TRUE

MISCELLANEOUS OPERATORS

There are also two miscellaneous operators worth mentioning.

Operator	Description
:	creates a series of numbers in sequence
%in%	identifies if an element belongs to a vector

The colon operator `(:)` is used to create a series of numbers in sequence. And the in operator `(%in%)` is used to identify if an element belongs to a vector. Let's look at an example for each to better understand what they do.

Here we assign the numbers 1 through 10 to an object named numbers.

```
numbers <- 1:10
```

And here we are creating a new dataframe, disc1, from an existing dataframe, reason, and keeping records where values of DCSREAS are "Death" or "Lost to Follow-up."

```
disc1 <- reason %>%
  filter(DCSREAS %in% c("Death","Lost to Follow-up"))
```

USUBJID	DCSREAS
01-351-711	Death
01-353-596	Lost to Follow-up
01-354-945	Death
01-355-531	Lost to Follow-up

DATA STEP OPTIONS AND STATEMENTS

SET

The SET statement in SAS® reads observations from one or more datasets. Here we create a new dataset, adsl2 from an existing dataset, adsl. Since we did nothing to the new dataset, adsl2 is exactly the same as adsl.

```
data adsl2;
  set adsl;
run;
```

In R we can get the same results by assigning adsl, an existing dataframe, to a new dataframe named adsl2.The assignment operator `(<-)` is the less than symbol followed by a dash.

```
adsl2 <- adsl
```

Example: when we have more than one dataframe

death:

USUBJID	AGE	DCSREAS
01-351-711	78	Death
01-354-945	77	Death

lost:

USUBJID	SEX	DCSREAS
01-353-596	F	Lost to Follow-up
01-355-531	F	Lost to Follow-up

We can use the SET statement to stack the death dataset and the lost dataset.

```
data all;
  set death lost;
run;
```

The results have all the observations from death, followed by all the observations from lost. Notice the death dataset did not have the variable SEX, so the values for SEX are missing in the death rows. Similarly lost did not have the variable AGE, so the values for AGE are missing in the lost rows.

USUBJID	AGE	DCSREAS	SEX
01-351-711	78	Death	
01-354-945	77	Death	
01-353-596	.	Lost to Follow-Up	F
01-355-531	.	Lost to Follow-Up	F

To do this in R you can use the `bind_rows()` function from the `dplyr` package. The `bind_rows()` function stacks rows of two or more dataframes. When row-binding, columns are matched by name, and any missing columns will be filled with `NA`.

```
all <- bind_rows(death,lost)
```

MERGE

The SAS® MERGE statement merges observations from two or more datasets. In R, there are a variety of join functions available, matching rows based on keys specified in the by parameter.

Consider these 2 datasets.

death:

USUBJID	AGE	DCSREAS
01-351-711	78	Death
01-354-945	77	Death

no_rel:

USUBJID	AEDECOD	AEREL
01-351-515	DIARRHOEA	REMOTE
01-351-711	SUDDEN DEATH	NONE

We can use the MERGE statement to merge observations from the death and no_rel datasets into a single observation in a new dataset, based on the value of USUBJID, the variable specified in the BY statement.

```
proc sort data=death;
  by usubjid;
run;

proc sort data=no_rel;
  by usubjid;
run;

data all;
  merge death no_rel;
  by usubjid;
run;
```

In the output the observations that are in both datasets have the variables from both datasets populated, and the observations that are not in both datasets have only the variables from the original dataset populated.

USUBJID	AGE	DCSREAS	AEDECOD	AEREL
01-351-515	.		DIARRHOEA	REMOTE
01-351-711	78	Death	SUDDEN DEATH	NONE
01-354-945	77	Death		

Note: The datasets being merged must be sorted by the BY variable.

To do this in R you can use the `full_join()` function from the `dplyr` package and specify USUBJID in the by parameter.

```
all <- full_join(death, no_rel, by='USUBJID')
```

In the join functions you provide a left side dataframe and a right side dataframe. The left is the first parameter, and the right is the second. The `full_join()` function adds columns from the right dataframe to the left dataframe and includes all rows in either dataframe. When using the join functions, rows are matched by keys, and any missing columns will be filled with `NA`.

As you can see the results are similar to the SAS® results but do differ slightly.

USUBJID	AGE	DCSREAS	AEDECOD	AEREL
01-351-711	78	Death	SUDDEN DEATH	NONE
01-354-945	77	Death	NA	NA
01-351-515	NA	NA	DIARRHOEA	REMOTE

First, SAS® and R represent missings differently. In R they are represented with the symbol `NA`. Second, the results are sorted differently. In SAS® you had to sort by the BY variable, USUBJID, prior to performing the merge, so the results are sorted by USUBJID. However, the join functions add the records from the right dataframe, no_rel, to the left dataframe, death. So all the rows from death appear first, followed by the rows that are only in no_rel.

To get the same order in R you can simply use the `arrange()` function from the `dplyr` package to arrange the resulting dataframe by USUBJID.

```
all <- full_join(death, no_rel, by='USUBJID') %>%
  arrange(USUBJID)
```

USUBJID	AGE	DCSREAS	AEDECOD	AEREL
01-351-515	NA	NA	DIARRHOEA	REMOTE
01-351-711	78	Death	SUDDEN DEATH	NONE
01-354-945	77	Death	NA	NA

What if we only wanted to keep records that are in both datasets?We can do this in R using the `inner_join()` function from the `dplyr` package and specifying USUBJID in the by parameter. The inner_join() function adds columns from the right dataframe to the left dataframe and includes all rows that are in both.

```
both <- inner_join(death, no_rel, by='USUBJID')
```

USUBJID	AGE	DCSREAS	AEDECOD	AEREL
01-351-711	78	Death	SUDDEN DEATH	NONE

We can also merge the data and keep only records from the death dataset.To do this in R we can use the `left_join()` function from the `dplyr` package and specify USUBJID in the by parameter. The left_join() function adds columns from the right dataframe to the left dataframe and includes all rows in the left dataframe.

```
in_death <- left_join(death, no_rel, by='USUBJID')
```

USUBJID	AGE	DCSREAS	AEDECOD	AEREL
01-351-711	78	Death	SUDDEN DEATH	NONE
01-354-945	77	Death	NA	NA

As you might have guessed we can also merge the data and keep only records from the no_rel dataset.In R we can use the `right_join()` function from the `dplyr` package and specify USUBJID in the by parameter. The right_join() function adds columns from the right dataframe to the left dataframe and includes all rows in the right dataframe. Just like our full_join() function example, we could use the `arrange()` function to sort the R results to match the SAS® results.

DROP/DROP=

In SAS® there is a DROP statement and a DROP= option. Although they process a bit differently, they both produce the same results and drop the variables specified.

Take a look at this dataset.

adsl:

USUBJID	TRT01A	TRT01AN	AGE	AGEU
01-351-515	Placebo	0	65	YEARS
01-351-528	Miracle High Dose	9	73	YEARS

We can use the DROP statement or the DROP= option to drop the AGE variable. Both methods produce the exact same results.

```
data adsl1;
   set adsl;
   drop age;
run;

data adsl1;
   set adsl (drop=age);
run;
```

To get the same results we can use the `select()` function from the `dplyr` package. The `select()` function can be used to select columns in a dataframe, and we can specify what columns to drop by preceding the column name with a dash `(-)`.

```
adsl1 <- adsl %>%
   select(-AGE)
```

USUBJID	TRT01A	TRT01AN	AGEU
01-351-515	Placebo	0	YEARS
01-351-528	Miracle High Dose	9	YEARS

KEEP/KEEP=

SAS® has both a KEEP statement and a KEEP= option that process a little differently but ultimately produce the same results and keep the variables specified. Here is an example of a simple dataset.

adsl:

USUBJID	TRT01A	TRT01AN	AGE	AGEU
01-351-515	Placebo	0	65	YEARS
01-351-528	Miracle High Dose	9	73	YEARS

We can use the KEEP statement or the KEEP= option to keep the USUBJID and AGE variables. Both methods produce the exact same results.

```
data age;
   set adsl;
   keep usubjid age;
run;

data age;
   set adsl (keep=usubjid age);
run;
```

To do this in R we can use the select() function. The select() function from the dplyr package can be used to select columns in a dataframe. Here we specify multiple columns to keep by separating their names with a comma.

```
age <- adsl %>%
   select(USUBJID, AGE)
```

USUBJID	AGE
01-351-515	65
01-351-528	73

Note: the `select()` function orders the columns based on the order in which you specify them.

RENAME/RENAME=

SAS® has both a RENAME statement and a RENAME= option that process a little differently but ultimately produce the same results and rename the variables specified.

Consider this dataset.

adsl:

USUBJID	TRT01A	TRT01AN	AGE	AGEU
01-351-515	Placebo	0	65	YEARS
01-351-528	Miracle High Dose	9	73	YEARS

We can use the RENAME statement or the RENAME= option to rename the AGE variable as AGE2. Both methods produce the exact same results.

```
data adsl1;
  set adsl;
  rename age=age2;
run;

data adsl1;
  set adsl((rename=(age=age2));
run;
```

To do this in R we can use the `rename()` function. The `rename()` function from the `dplyr` package changes the names of individual columns using `new_name = old_name` syntax.

```
adsl1 <- adsl %>%
  rename(AGE2=AGE)
```

USUBJID	TRT01A	TRT01AN	AGE2	AGEU
01-351-515	Placebo	0	65	YEARS
01-351-528	Miracle High Dose	9	73	YEARS

WHERE/WHERE=

In SAS® there is a WHERE statement and a WHERE= option. Although they process a bit differently, they both produce the same results and select observations that meet specified conditions.

Consider this dataset.

disc1:

USUBJID	DCSREAS
01-351-711	Death
01-353-596	Lost to Follow-up
01-354-945	Death
01-355-531	Lost to Follow-up

We can use the WHERE statement or the WHERE= option to select the observations where DCSREAS equals “Death.” Both methods produce the exact same results.

```
data death;
  set disc1;
  where dcsreas='Death';
run;

data death;
  set disc1 (where=(dcsreas='Death'));
run;
```

In R, we can get the same results using the filter() function. The filter() function from the dplyr package can be used to subset a dataframe, retaining all rows that satisfy your conditions. Recall the double equal operator (==) is used to check if each element of the first vector is equal to the corresponding element of the second vector. Here we are checking whether DCSREAS equals the text string “Death.”

```
death <- disc1 %>%
  filter(DCSREAS == "Death")
```

USUBJID	DCSREAS
01-351-711	Death
01-354-945	Death

PROCEDURES

PROC CONTENTS

In SAS®, PROC CONTENTS shows metadata about a dataset.

In R, there is no exact equivalent however the generic `str()` function gives us some similar information. The str() function compactly displays the internal structure of an R object.

Here we use PROC CONTENTS to get the contents of adsl.

```
proc contents data=adsl varnum;
run;
```

The output shows us general information such as the number of observations and the number of variables. Then for each variable we can see the variable type, length, format, and label.The results can be put into a dataset using the OUT= option.

```
proc contents data=adsl varnum out=adsl_contents;
run;
```

In R we can get some of this information by using the str() function.The output contains: the number of rows and columns, the column name, the column type, and the column label.

But note that column labels are not very typical of R, and mostly tend to exist when you're working with SAS® datasets or set a 'label' attribute to the column yourself.

```
adsl %>%
   str()
```

Although there's not currently a common function in R that produces the same results as PROC CONTENTS, there are ways that you can obtain the same results.

The `class()` function from base R tells you the type of object being inspected. This will give us similar information to the Type column from the PROC CONTENTS output.

Labels in R are just attributes of the columns within a dataframe.

The `attr()` function from base R allows you to access object attributes and get the value. But remember, labels in R are not a commonly used feature and are most prevalent when using SAS® data.

To replicate the results of PROC CONTENTS more closely, a function can be written that incorporates both of these functions and puts the results in a dataframe that can be printed.

PROC FREQ

In SAS®, PROC FREQ produces one-way to n-way frequency and cross-tabulation tables. This allows you to count the number of discrete values, or combinations of values, within a group of variables.

In R, the count() and group_by() functions from the dplyr package allow us to do the same thing.

The count() function counts the unique values of one or more columns.And the group_by() function performs operations within a specified group of columns.

TRT01P	SEX	ETHNIC	n
Miracle High Dose	F	NOT HISPANIC OR LATINO	3
Miracle High Dose	M	NOT HISPANIC OR LATINO	3
Miracle Low Dose	F	NOT HISPANIC OR LATINO	2
Miracle Low Dose	M	NOT HISPANIC OR LATINO	3
Placebo	F	HISPANIC OR LATINO	1
Placebo	F	NOT HISPANIC OR LATINO	1
Placebo	M	NOT HISPANIC OR LATINO	2

```r
by_count <- adsl %>%
  group_by(TRT01P) %>%
  count(SEX)
```

PROC MEANS

In SAS®, PROC MEANS provides data summarization tools to compute descriptive statistics for variables across all observations and within groups of observations.

In R, the summarize() and group_by() functions from the dplyr package, and a variety of available summary functions allow us to do the same thing.

The summarize() function summarizes data into a single row of values. The function uses summary functions that take a vector of values and return a single value.

And the group_by() function performs operations within a specified group of columns.When the data is grouped using the group_by() function, the summarize() function collapses each group into a single-row summary by applying summary functions to each group.

```r
by_stats <- adsl %>%
   group_by(TRT01P) %>%
   summarize(n=n(), mean=mean(AGE), sd=sd(AGE), q1=quantile(AGE, 0.25),
             q3=quantile(AGE, 0.75), nmissing=sum(is.na(AGE)))
```

TRT01P	n	mean	sd	q1	q3	nmissing
Miracle High Dose	6	71.5	10.578280	62.50	78.50	0
Miracle Low Dose	5	79.4	6.387488	76.00	83.00	0
Placebo	4	71.5	14.708274	62.25	82.25	0

Some commonly used summary functions are:

- n() from the dplyr package counts the number of values in a vector.
- quantile() returns sample quantiles corresponding to the given probabilities.*
- IQR() returns the interquartile range of a vector.*
- min() returns the minimum value in a vector.
- max() returns the maximum value in a vector.

- mean() returns the mean value of a vector.
- median() returns the median value of a vector.
- var() returns the variance of a vector.
- sd() returns the standard deviation of a vector.
- sum() returns the sum of all the values present in a vector.

PROC PRINT

In SAS®, PROC PRINT prints the observations in a dataset using all or some of the variables.

In R, there is no one function that is equivalent to PROC PRINT. R is generally able to write objects out to the console, which is inherently similar to PROC PRINT. For dataframes, functions can be used to specify the columns and observations to print.

Example: we use PROC PRINT to print the contents of adsl. Here we are looking at only part of the output from PROC PRINT. The full output shows all of the variables from adsl and all the values of those variables.

In R we can get the same results by simply sending the dataframe object to the console. The main difference with this output, and the PROC PRINT output, is that PROC PRINT shows all the variables and their values, however R lists out each of the columns, but only shows the values that fit.

Note, the exact same results can be produced using the print() function from base R.

PROC SORT

In SAS®, PROC SORT orders SAS® dataset observations by the values of one or more character or numeric variables.

In R, the arrange() function from the dplyr package allows us to do the same thing. The arrange() function orders the rows of a dataframe by the values of selected columns.

In R we can get the same results by specifying SEX as the input to the arrange() function.

```
adsl <- adsl %>%
  arrange(SEX)
```

PROC TRANSPOSE

In SAS®, PROC TRANSPOSE creates an output dataset by restructuring the values in a dataset, transposing selected variables into observations.

In R, the pivot_wider() and pivot_longer() functions from the tidyr package allow us to do the same thing.

The pivot_wider() function widens data, increasing the number of columns and decreasing the number of rows.And the pivot_longer() function lengthens data, increasing the number of rows and decreasing the number of columns.

Let's take a look at SAS® and R code to see how the functions can be used to produce the same results as PROC TRANSPOSE.Here is an example of a simple dataset. We want the values of PARAMCD to become columns containing the values of AVAL.

PARAMCD	AVAL
ALT	27
AST	40

To do this in SAS® we use PROC TRANSPOSE to transpose the data and output the transposed data as tadlbc.

```
proc transpose data=adlbc out=tadlbc;
  id paramcd;
  var aval;
run;
```

In R we can get the same results using the pivot_wider() function. The names_from parameter describes which column, or columns to get the name of the output column from, and the values_from parameter specifies which column, or columns, to get the cell values from.

```
tadlbc <- adlbc %>%
   pivot_wider(names_from = 'PARAMCD', values_from = 'AVAL')
```

ALT	AST
27	40

CONCLUSION

As the use of R continues to rise in clinical research, the need for clinical programmers to be proficient in both SAS and R is becoming increasingly important. The cheat sheet presented in this paper offers a practical and accessible tool for facilitating the transition between these two programming languages. By providing clear, side-by-side comparisons of SAS and R code, the cheat sheet empowers clinical programmers to efficiently navigate dual programming environments, thereby enhancing their ability to contribute to modern clinical research workflows.

REFERENCES

Atorus Academy — Open-source in Life Science Made Easy

ACKNOWLEDGMENTS

The authors acknowledge the open-source pharmaverse community for developing and maintaining validated tools that enable regulatory-compliant clinical programming in R.

CONTACT INFORMATION

(In case a reader wants to get in touch with you, please put your contact information at the end of the paper.)
Your comments and questions are valued and encouraged. Contact the author at:
Author Name: Lekitha JK
Company: Atorus Research
Address:
Work Phone: +91-7204363110
Email: Lekitha.jk@atorusresearch.com / academy@atorusreseach.com

.