**CHOP Data Analytics R Standards  
Trailblazers Club**

With R becoming a widely used tool within the team, there is a need to standardize the R coding process to allow for effective and efficient code review, project handoff, and project sustainment. CHOP will be adopting the following standards related to querying philosophy, organization, code structure, and code style for projects using R scripts and R markdowns.

**CHOP R Standards: Querying Philosophy**

Netezza will remain the standard tool for querying data and creating indicators that are used in metric building. Netezza is the fastest querying tool available and allows for the inclusion of all relevant information within the data mart in order to maintain a single source of truth for cohort and metric definition. The Netezza SQL code should include all of the relevant information in “data mart style”. Using the SQL-based complete dataset, it is recommended that analysts use R to manipulate, analyze, and visualize data.

**CHOP R Standards: Organization with R Projects**

**R Project Basics**

An R project is a way of easily organizing all of your R scripts by storing settings specific to that project. Scripts can then be run in the future if those settings change outside of your project (ex. New package version) and will also run for collaborators on their computer (i.e. code reviewers).

R projects will store all of your settings in a .Rproj file (i.e “foofy.Rproj”). This creates everything that the project needs in its own workspace or folder and touches *nothing that it did not create*; every time you open the .Rproj file you open a fresh instance of R studio with all of the settings for that specific project folder. R projects store the following settings:

* + 1. All code and outputs in one set location so you no longer need to set your working directory.
    2. Relative file paths for better reproducibility
    3. A clean R environment so that you no longer need to remove objects from other analyses before running your scripts (i.e. no more rm(list = ls())!)

R projects are superior to setwd() for the following reasons:

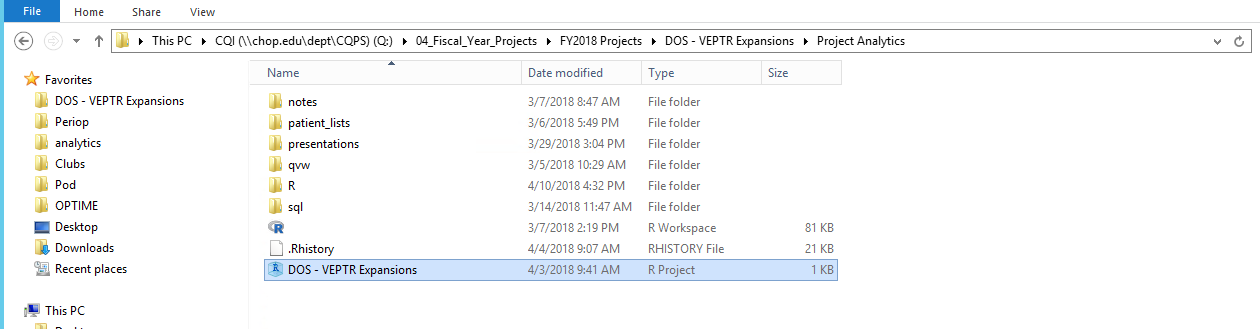
1. R projects allow for file paths to work for any analyst, whereas setwd()only allow file paths to work for the author
2. R projects eliminate the need to change file paths that may change.
3. R projects are better for analyst flow than the setwd() function (<https://www.tidyverse.org/articles/2017/12/workflow-vs-script/>)

For these reasons, the CHOP standard will be to *not* use setwd().

**Setting up an R Project**

In order for R projects to work, you will need to organize your R scripts/Rmd’s into directories. A directory is a folder that organizes all files needed for a given script that you use in your R script. This includes the R script itself as well as csv’s, SQL files, and/or imaging files.

To set up an R project, save your R project at the top level of your directory. When R projects are saved at the **top level** of your directory, R scripts are told to look for those files within that directory. This is how the set-up would work to save your R project at the **top level** of the directory:



As you can see, there is now an R project file saved within the directory (last file listed above). If you wanted to read in a csv from patient\_lists, your read.csv statement in your script would look like this:



Having these simple file paths within your R script allow for better reproducibility and better publishing capabilities via R Studio Connect since the script is no longer dependent on the full file path. Therefore, if something changes upstream in the file path (i.e. you rename your “FY2018Projects” folder “FY 2018 Projects”), your R scripts will not be impacted because they are not dependent on the “FY2018Projects” naming convention.

**Switching Between R Projects**

In order to make edits to an R script that does not belong to the current project that you are in, simply rick click on the R studio desktop icon, click “R Studio”, and a new window will open up. In this new session, you can then open up the project where the R script that you want to edit belongs and make the necessary changes.

**Additional Resource**

<https://www.youtube.com/watch?v=hKoSJGWnFFA>

**CHOP R Standards: Code Structure**

Everyone’s R code structure should also follow a similar structure. In this structure, R script/R markdowns will be organized into code sections and code chunks, and the order of these chunks will be consistent across all analysts.

***Creating and naming code sections and code chunks***

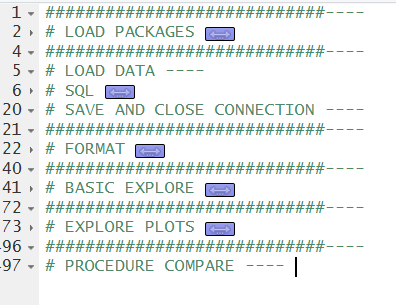
Code should be organized into sections and chunks. The name of a chunk/section should be short and informative. The name should tell the “what” and within each section the comments should explain the “why”.

To create a section in your .R file, add an short informative header as a comment, followed by at least 4 dashed lines: ---- . This will create a little arrow on the left hand side allowing you to collapse your code. You can also use the following short-cut to create sections: Ctrl+Shift+R

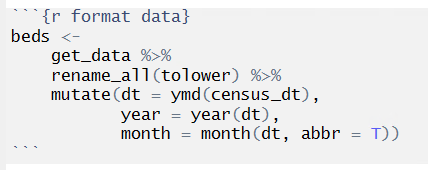
Machine generated alternative text:
1. 
# LOAD PACKAGES 
4 
6 
7 
8 
library(lubridate 
libraw(dplyr 
library( 
scales 
1 i braw ( ggp10t2 
library(ggthemes 
library( 
scales 

Machine generated alternative text:
1. 
# LOAD PACKAGES 

**Tip:** Placing several "#" above your section makes things look more readable when all your sections are collapsed:



To create a new code chunk in an R notebook or R Markdown file, press Ctrl + Alt + i. To name this chunk, type right after the "r" in the newly created chunk. Do not put a comma after the "r"



You can use the navigation in the lower left hand corner to efficiency navigate your code chunks

Machine generated alternative text:
Periop Strategic Dashboard 
Chunk I: load_packages 
Chunk 2 format data 
Financial Performance 
Chunk 3: sidebar fin 
Chunk 4: plots_fin 
Voice of the Customer 
Chunk 5: sidebar cust 
e Chunk 3: sidebar_fin : 
de: 
fill 
mdy 
yea 
cal 
01 ul 
s, 

**Tip:** ALT-L will collapse a chunk and ALT-SHIFT-L will expand it again

***Organizing Code Chunks***

*Code Chunk #1:*

The first chunk in your code should only include code that sets up the rest of your scripts. This includes loading packages and defining connection strings. You **should not** load packages or define connection strings in any other section of your code. This will allow you and/or a reviewer to know upfront what packages you are using and where you are pulling data from. Additionally, this will enable your R script to have all of the setting it needs for future runs before performing any sort of analysis. You can also use this package to automatically move package calls to the top of your script if you so choose:

<https://github.com/MilesMcBain/packup>.

Below is an example of what this first chunk may look like:



*Code Chunk #2:*

The second code chunk is where you will be bringing your data into your R environment. This can be performed via a SQL statement, read.csv statement, etc.

*Code Chunk #3:*

The third code chunk is where you should format all of your data that was pulled in code chunk #2 (i.e. case changes from UPPER to lower, date formatting, etc.)

*Subsequent Code Chunks:*

Once you have pulled in all of your data and formatted it appropriately, all subsequent code chunks should be broken up in your analysis into logical and consistent sections. Just like a new idea begins a new paragraph, your code should be broken up in the same manner and should be appropriately named.

**CHOP R Standards: Code Style**

**General Styling**

Analysts should use tidyverse R style in all coding, as this is the gold standard for R coding. This includes naming, assignment, spacing, and commenting.

Hadley Wickham’s tidyverse style guide: (<http://style.tidyverse.org/>)

Google’s R style guide: style (<https://google.github.io/styleguide/Rguide.xml>)

**Standard Packages**

A standard package is a package that has been determined to be the best package available to perform certain operations. The list of standard packages should encompass the vast majority of common operations that analysts do in typical analyses. Although analysts can technically install non-standard packages locally (i.e. not on Marx), this should only be done if an analyst is exploring a new package, if the package in question performs a needed operation that is not covered by a standard package, or if said package performs the same operation better than a standard package. In the latter 2 scenarios, the analyst in question must justify the use of this package in the analysis and also consider requesting that the package in question become standard. After a new package becomes a standard package, the package will be installed on Marx and educational materials will be provided to teach analysts how to use this new package.

All deliverables that are hosted on a prod server should use standard packages unless there is a reasonable justification for using a package outside of this standard list.

Below is a list of these standard packages.

|  |  |  |
| --- | --- | --- |
| Package | Description | Link |
| tidyverse | Contains a number of different packages that are useful for data manipulation (dplyr, lubridate, ggplot2, tidyr, etc.) | <https://www.r-pkg.org/pkg/tidyverse> |
| devtools | Allows installment of packages from github | <https://www.r-pkg.org/pkg/devtools> |
| rmarkdown | Create Rmarkdowns | <https://www.r-pkg.org/pkg/rmarkdown> |
| odbc | Connects to the CDW | <https://www.r-pkg.org/pkg/odbc> |
| REDCapR | Interacts with REDCap | <https://www.r-pkg.org/pkg/REDCapR> |
| sendmailR | Sends emails from R | <https://www.r-pkg.org/pkg/sendmailR> |
| EasyHTMLReport | Sends HTML reports to clients | <https://www.r-pkg.org/pkg/EasyHTMLReport> |
| htmlTable | Creates HTML tables | <https://www.r-pkg.org/pkg/htmlTable> |
| readxl | Reads excel files | <https://www.r-pkg.org/pkg/readxl> |
| writexl | Writes excel files | <https://www.r-pkg.org/pkg/writexl> |
| rocqi | Easily creates spc charts and performs other common functions specific for analysts | <https://github.research.chop.edu/pages/CQI/pkg-spc/rocqi/docs/reference/> |
| lintr | Reformats code to adhere to CHOP standards | <https://cran.r-project.org/web/packages/lintr/index.html> |
| data.table | Reformats data frames easily for publishing | <https://cran.r-project.org/web/packages/data.table/data.table.pdf> |
| highcharter | Creates interactive graphs for data visualization deliverables | <https://cran.r-project.org/web/packages/highcharter/highcharter.pdf> |

Justification must be provided by an analyst if he/she uses a package not contained within this list. This non-standard package may be used if it performs a different function than the packages previously listed. If an analyst would like to a new package to become a standard package, he/she should reach out to Christian Minich, Matt Dye, or Emily Schriver to begin the process.

**Naming**

*Functions***:** All functions outside of dplyr, gpplot2, tidyr, knitr, odbc, and readr will be called using the format “package\_name::function\_name”. This will make it easier for others to know what packages you’re using when employing certain functions.

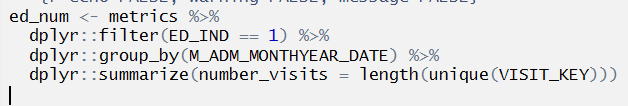
Example: reshape2::melt()

*Objects***:** all objects will be names using lowercase letters and underscores ( \_ ) rather than dots (.) or hyphens (-) to separate words in object names. Object/column names should be succinct and informative.

Example: meds\_raw, vanc\_ind

*Argument***:** If a function call is too long to fit on a single line, use one line each for the function name, each argument, and the closing “)”. Each new line should be indented by two spaces.

Example:



**Assignment**

When assigning values to new objects or variables, use “ <- “ rather than “ = ”. Be sure to include spaces before and after the symbols.

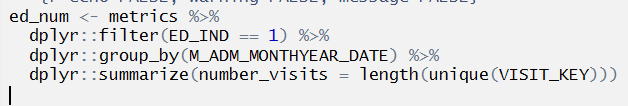
Exception: within dplyr::mutate() – in this argument, you should use “ = “ instead of “ <- “ since this is required by the function itself.

**Spacing and Indenting**

*Operators (+, =, -, etc.)*: all operators should be preceded with and followed by a space. Exceptions to the this rule include the following operators: ^, ::,:,!

*Pipes:* each function you pipe (%>%) should be on its own line (the pipe should end the line so that your code can run automatically). Additionally, each new line (new function being called by a pipe in this case) should be indented by two spaces. R generally does this automatically for you.

Example:



*Line Length:* lines should be no longer than 80 characters. Longer lines should be broken into two lines.

**Commenting Philosophy**

In general, comments should be used as a supplement to well-structured and readable code. In many instances, comments may not be necessary if you follow the standards listed above in terms of structure and naming conventions. Even with readable code, however, there are many times where comments should be used to clarify why you are doing what you are doing. This will be helpful to future you and other analysts. Comments should also be used to describe what you are doing if it is not clear.

When considering whether or not you should comment, you can ask yourself the following questions:

* “Would I know what is going on with this code in 3-6 months without comments?”
* “Would another analyst know what is happening in this code?”

If the answer is yes to either questions, a comment is likely needed. Remember, these comments should be used to answer “why” questions rather than “what” questions. For example, instead of commenting “I am filtering this dataset” the comment should read “I am filtering this dataset because…”

When commenting within a code chunk, you should always have a space after the ‘#’ to signal to others that this is a comment. Commented out code, however, should not have a space after the ‘#’.

* Comments: # with a space for comments
* Commented out code: #with no space for commented out code (code that you are not running)

Do some of the spacing and naming rules seem confusing or difficult to implement? Fear not! You can use the lintr package to style check your code. Using the following code will automatically check your code style:

library(lintr)

lintr::lint(“H:/project\_folder/project\_name.R’)

**Code that should NOT be used:**

* Require()
* Attach()
* Setwd()
* .libPaths(‘H:/RLIBS’)
* Semicolon (;)

**CHOP R Standards: Publishing with R Studio Connect**

All applications that live on R Studio Connect should be sourced from a data mart in order to improve efficiency. Additionally, the various templates and guides available on chopr should be used when created applications in order to adhere to UI standards.