Introduction to R and the tidyverse

Kim Dill-McFarland, kadm@uw.edu

version March 25, 2020

Contents

Overview

In this workshop, we introduce you to R and RStudio at the beginner level as well as begin to work in the tidyverse. In it, we cover:

- R and RStudio including projects, scripts, and packages
- Reading in data as a data frame
- Vectors, single values, and data types
- The help function
- Manipulating data frames with tidyverse verbs

We will do all of our work in RStudio. RStudio is an integrated development and analysis environment for R that brings a number of conveniences over using R in a terminal or other editing environments.

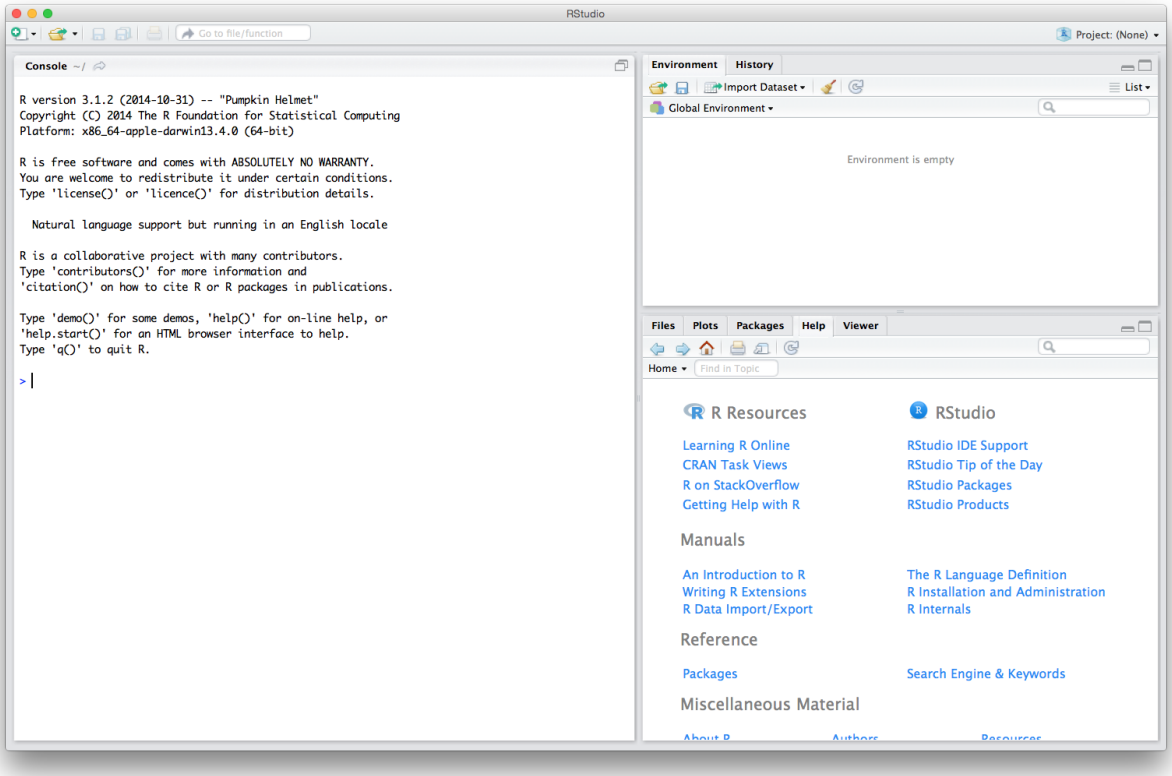
During the workshop, we will build an R script together, which will be posted as 'live_notes' after the workshop here.

Prior to the workshop

Please install R and RStudio. See the setup instructions for more details.

A Tour of RStudio

When you start RStudio, you will see something like the following window appear:



Notice that the window is divided into three "panes":

- Console (the entire left side): this is your view into the R engine. You can type in R commands here and see the output printed by R. (To make it easier to tell them apart, your input is printed in blue, while

the output is black.) There are several editing conveniences available: use up and down arrow keys to go back to previously entered commands, which can then be edited and re-run; TAB for completing the name before the cursor; see more in online docs.

- Environment/History (tabbed in upper right): view current user-defined objects and previously-entered commands, respectively.
- Files/Plots/Packages/Help (tabbed in lower right): as their names suggest, these are used to view the contents of the current directory, graphics created by the user, install packages, and view the built-in help pages.

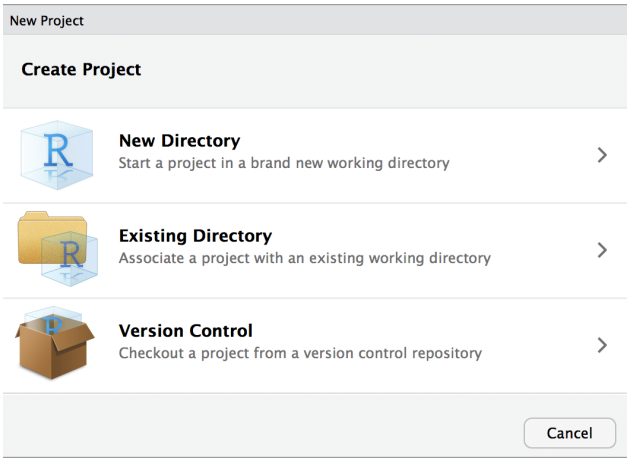
To change the look of RStudio, you can go to Tools -> Global Options -> Appearance and select colors, font size, etc. If you plan to be working for longer periods, we suggest choosing a dark background color scheme to save your computer battery and your eyes.

RStudio Projects

Projects are a great feature of RStudio. When you create a project, RStudio creates an `.Rproj` file that links all of your files and outputs to the project directory. When you import data, R automatically looks for the file in the project directory instead of you having to specify a full file path on your computer like `/Users/username/Desktop/`. R also automatically saves any output to the project directory. Finally, projects allow you to save your R environment in `.RData` so that when you close RStudio and then re-open it, you can start right where you left off without re-importing any data or re-calculating any intermediate steps.

RStudio has a simple interface to create and switch between projects, accessed from the button in the top-right corner of the RStudio window. (Labelled "Project: (None)", initially.)

Create a Project Let's create a project to work in for this workshop. Start by clicking the "Project" button in the upper right or going to the "File" menu. Select "New Project" and the following will appear.



You can either create a project in an existing directory or make a new directory on your computer - just be sure you know where it is.

After your project is created, navigate to its directory using your Finder/File explorer. You will see the `.RProj` file has been created.

To access this project in the future, simply double-click the `RProj` and RStudio will open the project or choose File > Open Project from within an already open RStudio window.

R Scripts

R script files are the primary way in which R facilitates reproducible research. They contain the code that loads your raw data, cleans it, performs the analyses, and creates and saves visualizations. R scripts maintain

a record of everything that is done to the raw data to reach the final result. That way, it is very easy to write up and communicate your methods because you have a document listing the precise steps you used to conduct your analyses. This is one of R's primary advantages compared to traditional tools like Excel, where it may be unclear how to reproduce the results.

Generally, if you are testing an operation (*e.g.* what would my data look like if I applied a log-transformation to it?), you should do it in the console (left pane of RStudio). If you are committing a step to your analysis (*e.g.* I want to apply a log-transformation to my data and then conduct the rest of my analyses on the log-transformed data), you should add it to your R script so that it is saved for future use.

Additionally, you should annotate your R scripts with comments. In each line of code, any text preceded by the `#` symbol will not execute. Comments can be useful to remind yourself and to tell other readers what a specific chunk of code does.

Let's create an R script (File > New File > R Script) and save it as `live_notes.R` in your main project directory. If you again look to the project directory on your computer, you will see `live_notes.R` is now saved there.

We will work together to create and populate the `live_notes.R` script throughout this workshop.

R packages

R packages are units of shareable code, containing functions that facilitate and enhance analyses. Let's install `ggplot2`, a very popular data visualization package in R that we will use later in the workshop. Packages are typically installed from CRAN (The Comprehensive R Archive Network), which is a database containing R itself as well as many R packages. Any package can be installed from CRAN using the `install.packages` function. You can input this into your console (as opposed to `live_notes.R`) since once a package is installed on your computer, you won't need to re-install it again.

```
install.packages("ggplot2")
```

After installing a package, and *every time* you open a new RStudio session, the packages you want to use need to be loaded into the R workspace with the `library` function. This tells R to access the package's functions and prevents RStudio from lags that would occur if it automatically loaded every downloaded package every time you opened it.

```
# Data visualization
library(ggplot2)
```

Getting started

Organize data

Create a directory called `data` and move the data files you received via email to this directory.

Loading data into an R data frame

One of R's most essential data structures is the data frame, which is simply a table of `m` columns by `n` rows. First, we will read in the RNA-seq cleaning metrics data into RStudio using the base R `read.table` function.

Each R function follows the following basic syntax, where `Function` is the name of the function.

```
Function(argument1=..., argument2=..., ...)
```

The read.table has many arguments; however, we only need to specify 3 arguments to correctly read in our data as a data frame. For our data, we will need to specify:

- `file` - gives the path to the file that we want to load from our working directory (current project directory).

- `sep` - tells R that our data are comma-separated
- `header` - tells R that the first row in our data contains the names of the variables (columns).

We will store the data as an *object* named `dat` using the assignment operator `<-`, so that we can re-use it in our analysis.

```
# read the data and save it as an object
dat <- read.table(file="data/AM.MDM.data.cleaning.metrics.csv",
                  sep=",", header=TRUE)
```

Now whenever we want to use these data, we simply call `dat`

Help function

You can read up about the different arguments of a specific function by typing `?Function` or `help(Function)` in your R console.

```
?read.table
```

You will notice that there are multiple functions of the `read.table` help page. This include similar and related functions with additional options. For example, since our data are in `.csv` format, we could've instead read them into R with `read.csv` which assumes the options `sep=",", header=TRUE` by default.

```
# read the data with different function
dat <- read.csv(file="data/AM.MDM.data.cleaning.metrics.csv")
```

Data types

This data frame consists of 24 rows (observations) and 55 columns (variables). You can see this quickly using the dimension function `dim`

```
dim(dat)
```

```
## [1] 24 55
```

Each column and each row of a data frame are individual R vectors. R vectors are one-dimensional arrays of data. For example, we can extract column vectors from data frames using the `$` operator.

```
# Extract the oxygen concentrations
dat$sampID
```

```
##  [1] AM10_AM_Media  AM10_AM_TB     AM10_MDM_Media AM10_MDM_TB    AM12_AM_Media
##  [6] AM12_AM_TB     AM12_MDM_Media AM12_MDM_TB    AM15_AM_Media  AM15_AM_TB
## [11] AM15_MDM_Media AM15_MDM_TB    AM17_AM_Media  AM17_AM_TB     AM17_MDM_Media
## [16] AM17_MDM_TB    AM18_AM_Media  AM18_AM_TB     AM18_MDM_Media AM18_MDM_TB
## [21] AM19_AM_Media  AM19_AM_TB     AM19_MDM_Media AM19_MDM_TB
## 24 Levels: AM10_AM_Media AM10_AM_TB AM10_MDM_Media ... AM19_MDM_TB
```

R objects have several different classes (types). Our data frame contains three R data types. The base R `class` function will tell you what data type an object is.

```
class(dat)
```

```
## [1] "data.frame"
```

```
class(dat$sampID)
```

```
## [1] "factor"
```

```
class(dat$raw)
```

```
## [1] "integer"
```

```
class(dat$PCT_PF_ALIGNED)
```

```
## [1] "numeric"
```

We see that our `sampID` column is a factor, meaning it is a non-numeric value with a set number of unique levels. Non-numeric data that don't have set levels are class `character`.

Then we have 2 types of numeric data. `integer` meaning a whole number and `numeric` meaning a number with decimal values.

There is one additional data types that you will commonly come across in R that is not in these data. This is the class `logical`, which is a TRUE/FALSE designation.

Working with vectors and data frames

Operating on vectors

A large proportion of R functions operate on vectors to perform quick computations over their values. Here are some examples:

```
# Compute the variance of raw sequence totals
var(dat$raw)
```

```
## [1] 4.260591e+13
```

```
# Find whether any samples have greater than 1 billion raw sequences
dat$raw > 1E9
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
# Convert PCT_PF_ALIGNED from decimal to percent
dat$PCT_PF_ALIGNED * 100
```

```
##  [1] 99.58188 99.60117 99.60300 99.59785 99.61403 99.61142 99.60669 99.59355
##  [9] 99.54444 99.53863 99.50553 99.52528 99.56393 99.57835 99.56958 99.57172
## [17] 99.62917 99.62982 99.61215 99.60768 99.60724 99.60417 99.58024 99.58072
```

```
# Find the unique values of sampID
unique(dat$sampID)
```

```
##  [1] AM10_AM_Media  AM10_AM_TB     AM10_MDM_Media AM10_MDM_TB    AM12_AM_Media
##  [6] AM12_AM_TB     AM12_MDM_Media AM12_MDM_TB    AM15_AM_Media  AM15_AM_TB
## [11] AM15_MDM_Media AM15_MDM_TB    AM17_AM_Media  AM17_AM_TB     AM17_MDM_Media
## [16] AM17_MDM_TB    AM18_AM_Media  AM18_AM_TB     AM18_MDM_Media AM18_MDM_TB
## [21] AM19_AM_Media  AM19_AM_TB     AM19_MDM_Media AM19_MDM_TB
## 24 Levels: AM10_AM_Media AM10_AM_TB AM10_MDM_Media ... AM19_MDM_TB
```

```
# Because this variable is a factor, we can also use
levels(dat$sampID)
```

```
##  [1] "AM10_AM_Media"  "AM10_AM_TB"     "AM10_MDM_Media" "AM10_MDM_TB"
##  [5] "AM12_AM_Media"  "AM12_AM_TB"     "AM12_MDM_Media" "AM12_MDM_TB"
##  [9] "AM15_AM_Media"  "AM15_AM_TB"     "AM15_MDM_Media" "AM15_MDM_TB"
## [13] "AM17_AM_Media"  "AM17_AM_TB"     "AM17_MDM_Media" "AM17_MDM_TB"
## [17] "AM18_AM_Media"  "AM18_AM_TB"     "AM18_MDM_Media" "AM18_MDM_TB"
## [21] "AM19_AM_Media"  "AM19_AM_TB"     "AM19_MDM_Media" "AM19_MDM_TB"
```

Using the correct class

Functions executed on an object in R may respond exclusively to one or more data types or may respond differently depending on the data type. For example, you cannot take the mean of a factor or character.

```
# Compute the mean of sampID
mean(dat$sampID)
```

```
## Warning in mean.default(dat$sampID): argument is not numeric or logical:
## returning NA
```

```
## [1] NA
```

Subsetting vectors and data frames

Since vectors are 1D arrays of a defined length, their individual values can be retrieved using vector indices. R uses 1-based indexing, meaning the first value in an R vector corresponds to the index 1. Each subsequent element increases the index by 1. For example, we can extract the value of the 5th element of the sampID vector using the square bracket operator `[]` like so.

```
dat$sampID[5]
```

```
## [1] AM12_AM_Media
## 24 Levels: AM10_AM_Media AM10_AM_TB AM10_MDM_Media ... AM19_MDM_TB
```

In contrast, data frames are 2D arrays so indexing is done across both dimensions as `[rows, columns]`. So, we can extract the same oxygen value directly from the data frame knowing it is in the 5th row and 1st column.

```
dat[5, 1]
```

```
## [1] AM12_AM_Media
## 24 Levels: AM10_AM_Media AM10_AM_TB AM10_MDM_Media ... AM19_MDM_TB
```

The square bracket operator is most often used with logical vectors (TRUE/FALSE) to subset data. For example, we can subset our data frame to all observations (rows) with greater than 100 million raw sequences.

```
# Create logical vector for which oxygen values are 0
logical.vector <- dat$raw > 100E6
#View vector
logical.vector
```

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

```
#Apply vector to data frame to select only observations where the logical vector is TRUE (i.e. the oxyg
dat[logical.vector, ]
```

```
##            sampID       raw      trim to.be.aligned secondary.align     align
## 15 AM17_MDM_Media 100377274 100175714     217381758       138913767 217381758
##      paired R1.paired R2.paired align.paired both.align.paired one.align.paired
## 15 78467991  39236111  39231880     78462492          78462492             5499
##    PCT_PF_ALIGNED   PF_BASES PF_ALIGNED_BASES CODING_BASES  UTR_BASES
## 15      0.9956958 3202698312       3188913364   1110796510 1152263169
##    INTRONIC_BASES INTERGENIC_BASES NUM_R1_TRANSCRIPT_STRAND_READS
## 15      400110478        525743207                          348427
##    NUM_R2_TRANSCRIPT_STRAND_READS NUM_UNEXPLAINED_READS
## 15                       22955629                196413
##    PCT_R1_TRANSCRIPT_STRAND_READS PCT_R2_TRANSCRIPT_STRAND_READS
## 15                       0.014951                       0.985049
```

```
##    PCT_CODING_BASES PCT_UTR_BASES PCT_INTRONIC_BASES PCT_INTERGENIC_BASES
## 15         0.348331      0.361334           0.125469             0.164866
##    PCT_MRNA_BASES PCT_USABLE_BASES MEDIAN_CV_COVERAGE MEDIAN_5PRIME_BIAS
## 15       0.709665          0.70661           0.825082           0.906945
##    MEDIAN_3PRIME_BIAS MEDIAN_5PRIME_TO_3PRIME_BIAS to.be.aligned_filter
## 15           0.120767                     5.836842             50872052
##    align_filter paired_filter R1.paired_filter R2.paired_filter
## 15     50872052      50872052         25437396         25434656
##    align.paired_filter both.align.paired_filter one.align.paired_filter
## 15            50868988                 50868988                   3064
##    to.be.aligned_filter.paired align_filter.paired paired_filter.paired
## 15                    50868988            50868988             50868988
##    R1.paired_filter.paired R2.paired_filter.paired align.paired_filter.paired
## 15                25434494                25434494                   50868988
##    both.align.paired_filter.paired Assigned Unassigned_NoFeatures
## 15                        50868988 19369147               4266648
##    Unassigned_Ambiguity Assigned_paired Unassigned_NoFeatures_paired
## 15              1801763        19366921                      4266029
##    Unassigned_Ambiguity_paired
## 15                     1801544
```

Subsetting is extremely useful when working with large data. We will learn more complex subsets next using the tidyverse. But first...

Quick reference: Conditional statements

Statement	Meaning
`<-`	Assign to object in environment
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to
`%in%`	In or within
`is.na()`	Is missing, *e.g* NA
`!is.na()`	Is not missing
`&`	And
`|`	Or

Exercises: Part 1

1. Install the `tidyverse` package.

*Please note that if you have **R v3.3 or older**, you may not be able to install `tidyverse`. In this case, you need to separately install each package within the tidyverse. This includes:* `readr`, `tibble`, `dplyr`, `tidyr`, `stringr`, `ggplot2`, `purr`, `forcats`

2. Using help to identify the necessary arguments for the log function compute the natural logarithm of 4, base 2 logarithm of 4, and base 4 logarithm of 4.
3. Using an R function, determine what data type the `paired` variable is.
4. Using indexing and the square bracket operator `[]`:
    - determine what `trim` value occurs in the 20th row

- return the cell where `raw` equals 95,004,980

5. Subset the data to observations where `Assigned` is greater than or equal to 20 million. *Hint*: Use a logical vector.

The tidyverse

What is a tidyverse?

The R tidyverse is a set of packages aimed at making, manipulating, and plotting tidy data. Everything we've done thus far has been in base R. Now we will move into the tidyverse!

First, we need to load the package, which will give us a message detailing all the packages this one command loads for us.

```
library(tidyverse)
```

```
## -- Attaching packages ----------------------------------------------- tidyverse 1.3.0 --
```

```
## v tibble  2.1.3     v dplyr   0.8.5
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
## v purrr   0.3.3
```

```
## -- Conflicts -------------------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Loading data with readr

The most common formats for medium to medium-large data sets are *comma-* or *tab-separated values* (`.csv` and `.tsv`/`.txt`, respectively). In this format, the data is stored in plain text, with one observation per line, and variables within each observation separated by a comma or tab.

The readr functions `read_csv` and `read_tsv` help read in data at quick speeds compared to base R's `read.csv` and `read.tsv` functions. Furthermore, readr's data loading functions automatically parse your data into data types (numeric, character, etc) based on the values in the first 1000 rows.

Let's start by re-loading in the data we previously loaded with base R's `read.table`.

```
dat <- read_csv(file="data/AM.MDM.data.cleaning.metrics.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   sampID = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

We can then view all the classes it automatically assigned to our variables.

```
spec(dat)
```

```
## cols(
##   sampID = col_character(),
##   raw = col_double(),
##   trim = col_double(),
##   to.be.aligned = col_double(),
##   secondary.align = col_double(),
```

```
##   align = col_double(),
##   paired = col_double(),
##   R1.paired = col_double(),
##   R2.paired = col_double(),
##   align.paired = col_double(),
##   both.align.paired = col_double(),
##   one.align.paired = col_double(),
##   PCT_PF_ALIGNED = col_double(),
##   PF_BASES = col_double(),
##   PF_ALIGNED_BASES = col_double(),
##   CODING_BASES = col_double(),
##   UTR_BASES = col_double(),
##   INTRONIC_BASES = col_double(),
##   INTERGENIC_BASES = col_double(),
##   NUM_R1_TRANSCRIPT_STRAND_READS = col_double(),
##   NUM_R2_TRANSCRIPT_STRAND_READS = col_double(),
##   NUM_UNEXPLAINED_READS = col_double(),
##   PCT_R1_TRANSCRIPT_STRAND_READS = col_double(),
##   PCT_R2_TRANSCRIPT_STRAND_READS = col_double(),
##   PCT_CODING_BASES = col_double(),
##   PCT_UTR_BASES = col_double(),
##   PCT_INTRONIC_BASES = col_double(),
##   PCT_INTERGENIC_BASES = col_double(),
##   PCT_MRNA_BASES = col_double(),
##   PCT_USABLE_BASES = col_double(),
##   MEDIAN_CV_COVERAGE = col_double(),
##   MEDIAN_5PRIME_BIAS = col_double(),
##   MEDIAN_3PRIME_BIAS = col_double(),
##   MEDIAN_5PRIME_TO_3PRIME_BIAS = col_double(),
##   to.be.aligned_filter = col_double(),
##   align_filter = col_double(),
##   paired_filter = col_double(),
##   R1.paired_filter = col_double(),
##   R2.paired_filter = col_double(),
##   align.paired_filter = col_double(),
##   both.align.paired_filter = col_double(),
##   one.align.paired_filter = col_double(),
##   to.be.aligned_filter.paired = col_double(),
##   align_filter.paired = col_double(),
##   paired_filter.paired = col_double(),
##   R1.paired_filter.paired = col_double(),
##   R2.paired_filter.paired = col_double(),
##   align.paired_filter.paired = col_double(),
##   both.align.paired_filter.paired = col_double(),
##   Assigned = col_double(),
##   Unassigned_NoFeatures = col_double(),
##   Unassigned_Ambiguity = col_double(),
##   Assigned_paired = col_double(),
##   Unassigned_NoFeatures_paired = col_double(),
##   Unassigned_Ambiguity_paired = col_double()
## )
```

First, you'll see that `sampID` is no longer a factor. What happened?? Tidyverse is more strict about types and classes than base R. Since we did not tell it that this column was a factor, it does not assume that it is.

Second, you'll see that all our numeric/integer values are now `double`. This is another number class in R that stands for "double precision floating point numbers". Under the hood, doubles are more exact than numeric and more flexible than integer. So, tidyverse preferentially assigns number data to double.

If we wanted to change any of these assignments, we could like so.

```
dat <- read_csv(file="data/AM.MDM.data.cleaning.metrics.csv",
                col_types=cols(sampID=col_factor()))
```

Data wrangling with dplyr

`dplyr` is a package within the tidyverse. It provides many functions for manipulating data frames including typical tasks like:

- `select` a subset of variables (columns)
- `filter` out a subset of observations (rows)
- `rename` variables
- `arrange` the observations by sorting a variable in ascending or descending order
- `mutate` all values of a variable (apply a transformation)
- `group_by` a variable and `summarise` data by the grouped variable
- `*_join` two data frames into a single data frame

and others...

While base R can accomplish all of these tasks, base R code is rather slow and can quickly become extremely convoluted.

Currently, the most popular alternative for data wrangling is the package *dplyr*. It is so good at what it does, and integrates so well with other popular tools like *ggplot2*, that it has rapidly become the de-facto standard and it is what we will focus on today.

Compared to base R, dplyr code runs much faster. It is also much more readable because all operations are based on using dplyr functions or *verbs* (select, filter, mutate...) rather than base R's more difficult to read indexing system (brackets, parentheses...).

Each verb works similarly:

- input data frame in the first argument
- other arguments can refer to variables as if they were local objects
- output is another data frame

Before working with our data, we first want to make a copy of it so that we may revert to it quickly if we make any mistakes. This is best practices for data science in general.

```
raw_data <- dat
```

Select

You can use the `select` function to focus on a subset of variables (columns). Let's select the variables that we will need for this workshop. Here, we will use `dat` and select the variables:

- `sampID` Sample ID
- `raw` Total raw sequences
- `trim` Total trimmed sequences
- `both.align.paired` Total sequences where both reads aligned
- `both.align.paired_filter` Total sequences where both reads aligned and were high-quality
- `Assigned_paired` Total sequences assigned to a known gene

```
dat <- select(dat,
              sampID, raw, trim,
```

```
              both.align.paired,
              both.align.paired_filter,
              Assigned_paired)
```

Filter

You can use `filter` to select specific rows based on a logical condition of a variable. Below we filter the data such that we only retain Media samples. Note that here we're using a conditional statement that you have not seen before. Check it out with `?grepl`

```
dat <- filter(dat, grepl("Media",sampID))
```

As we saw earlier, conditional statements and logical expressions in R are extremely powerful and allow us to filter the data in almost any way imaginable.

Piping with %>%

Recall the basic dplyr verb syntax:

- input data frame in the first argument
- other arguments can refer to variables as if they were local objects
- output is another data frame

Our data cleaning code continuously overwrites the `dat` object every time we call a dplyr verb. Instead, we can chain commands together using the pipe `%>%` operator. This works nicely to condense code and to improve readability.

`f(x) %>% g(y)` is the same as `g(f(x),y)`

`select(dat, raw)` is the same as `dat %>% select(raw)`

Let's return to our `raw_dat` and perform the select and filter steps with pipes. Note how I've added comments within the function to aid the reader.

```
dat <- raw_data %>%
  #Select variables of interest
  select(sampID, raw, trim,
         both.align.paired,
         both.align.paired_filter,
         Assigned_paired) %>%
  #Filter to Media samples only
  filter(grepl("Media",sampID))
```

Manipulating data frames with tidyr

We will now move into another tidyverse package! tidyr contains functions for manipulating entire data frames including

- `pivot_longer` convert wide to long format
- `pivot_wider` convert long to wide format

and others that we will not go over today.

Wide vs. long data

Wide and long describe two different formats for a data frame. Wide data is where each variable is given its own column. Narrow data is where one column contains all of the variable names, and another column contains all of the values of these variables.

For example, this wide data

```
##   sample_ID year_2015 year_2016 year_2017
## 1         1     0.288     1.140     1.051
## 2         2     0.788     0.246     0.957
## 3         3     0.409     0.728     1.457
## 4         4     0.883     1.092     0.953
```

contains the same information as this long data.

```
##    sample_ID      Year Value
## 1          1 year_2015 0.288
## 2          2 year_2015 0.788
## 3          3 year_2015 0.409
## 4          4 year_2015 0.883
## 5          1 year_2016 1.140
## 6          2 year_2016 0.246
## 7          3 year_2016 0.728
## 8          4 year_2016 1.092
## 9          1 year_2017 1.051
## 10         2 year_2017 0.957
## 11         3 year_2017 1.457
## 12         4 year_2017 0.953
```

Our data is in the wide format.

```
dim(dat)
```

```
## [1] 12  6
```

```
head(dat)
```

```
## # A tibble: 6 x 6
##   sampID         raw   trim both.align.pair~ both.align.paired_~ Assigned_paired
##   <fct>        <dbl>  <dbl>            <dbl>               <dbl>           <dbl>
## 1 AM10_AM_M~  8.61e7 8.59e7         81594792            51271616        19076534
## 2 AM10_MDM_~  8.38e7 8.37e7         79958504            52990234        19078230
## 3 AM12_AM_M~  7.89e7 7.89e7         75681538            52937020        20176916
## 4 AM12_MDM_~  8.57e7 8.56e7         81921290            57680356        21639973
## 5 AM15_AM_M~  8.26e7 8.24e7         78050428            50822840        19441996
## 6 AM15_MDM_~  1.00e8 9.96e7         93820234            56545658        21416597
```

pivot_

Here, we pivot our data from wide to long format, specifying that we want to pivot everything *except* the sample IDs.

```
dat <- dat %>%
  pivot_longer(-sampID, names_to = "name", values_to = "value")
```

This results in a much longer data frame

```
dim(dat)
```

```
## [1] 60  3
```

```
head(dat)
```

```
## # A tibble: 6 x 3
##   sampID         name                      value
```

```
##   <fct>          <chr>                     <dbl>
## 1 AM10_AM_Media  raw                     86128242
## 2 AM10_AM_Media  trim                    85926379
## 3 AM10_AM_Media  both.align.paired       81594792
## 4 AM10_AM_Media  both.align.paired_filter 51271616
## 5 AM10_AM_Media  Assigned_paired         19076534
## 6 AM10_MDM_Media raw                     83755236
```

We can then undo this with `pivot_wider`.

```
dat <- dat %>%
  pivot_wider(names_from = "name", values_from = "value")

head(dat)
```

```
## # A tibble: 6 x 6
##   sampID         raw   trim both.align.pair~ both.align.paired_~ Assigned_paired
##   <fct>        <dbl>  <dbl>            <dbl>               <dbl>           <dbl>
## 1 AM10_AM_M~  8.61e7 8.59e7         81594792            51271616        19076534
## 2 AM10_MDM_~  8.38e7 8.37e7         79958504            52990234        19078230
## 3 AM12_AM_M~  7.89e7 7.89e7         75681538            52937020        20176916
## 4 AM12_MDM_~  8.57e7 8.56e7         81921290            57680356        21639973
## 5 AM15_AM_M~  8.26e7 8.24e7         78050428            50822840        19441996
## 6 AM15_MDM_~  1.00e8 9.96e7         93820234            56545658        21416597
```

`pivot_` functions are often difficult to wrap your head around. Be sure to always look at the data before and after to make sure you've accomplished what you set out to do!

Graphics with ggplot2

ggplot2 is the tidyverse's main plotting package. Full documentation is available at docs.ggplot2.org

Why ggplot?

ggplot2 is an implementation of *Grammar of Graphics* (Wilkinson 1999) for R

Benefits:

- handsome default settings
- snap-together building block approach
- automatic legends, colors, facets
- statistical overlays: regressions lines and smoothers (with confidence intervals)

Drawbacks:

- it can be hard to get it to look *exactly* the way you want
- requires having the input data in a certain format

ggplot building blocks

- data: 2D table (`data.frame`) of *variables*
- *aesthetics*: map variables to visual attributes (e.g., position)
- *geoms*: graphical representation of data (points, lines, etc.)
- *stats*: statistical transformations to get from data to points in the plot(binning, summarizing, smoothing)
- *scales*: control *how* to map a variable to an aesthetic
- *facets*: juxtapose mini-plots of data subsets, split by variable(s)
- *guides*: axes, legend, etc. reflect the variables and their values

Idea: independently specify and combine the blocks to create the plot you want.

There are at least three things we have to specify to create a plot:

1. data
2. aesthetic mappings from data variables to visual properties
3. a layer describing how to draw those properties

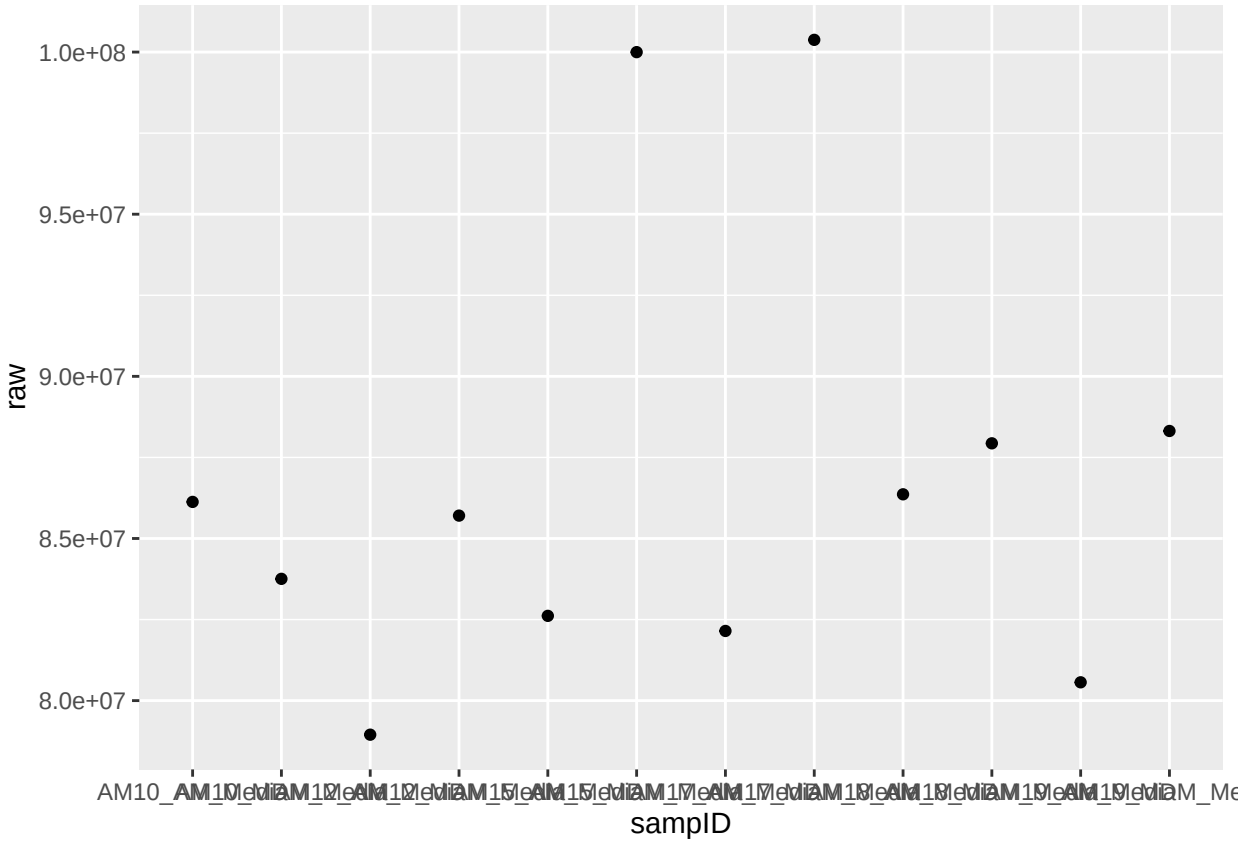
geom_point

The point geom is used to create scatterplots.

The first argument of ggplot is the data, hence we pipe our data into the ggplot function. The second argument is the aesthetics, which are used by all subsequent geoms. We then add a geom_point to the ggplot object, specifying that we want size 1 points. Here, geom_point takes the aesthetics of the ggplot object and understands that we want to plot `sampID` on the x-axis and `raw` on the y-axis.

Let's examine each sample's total raw sequences.

```
dat %>%
  ggplot(aes(x=sampID, y=raw)) +
  geom_point()
```



geom_bar

Unfortunately, we don't have enough time to go into all the amazing things ggplot can do. So, here we'll provide one intense ggplot statement and go through it step-by-step in an effort to get in as many useful functions as possible. We'll start with `raw_dat` to again demonstrate the dplyr verbs you learned earlier

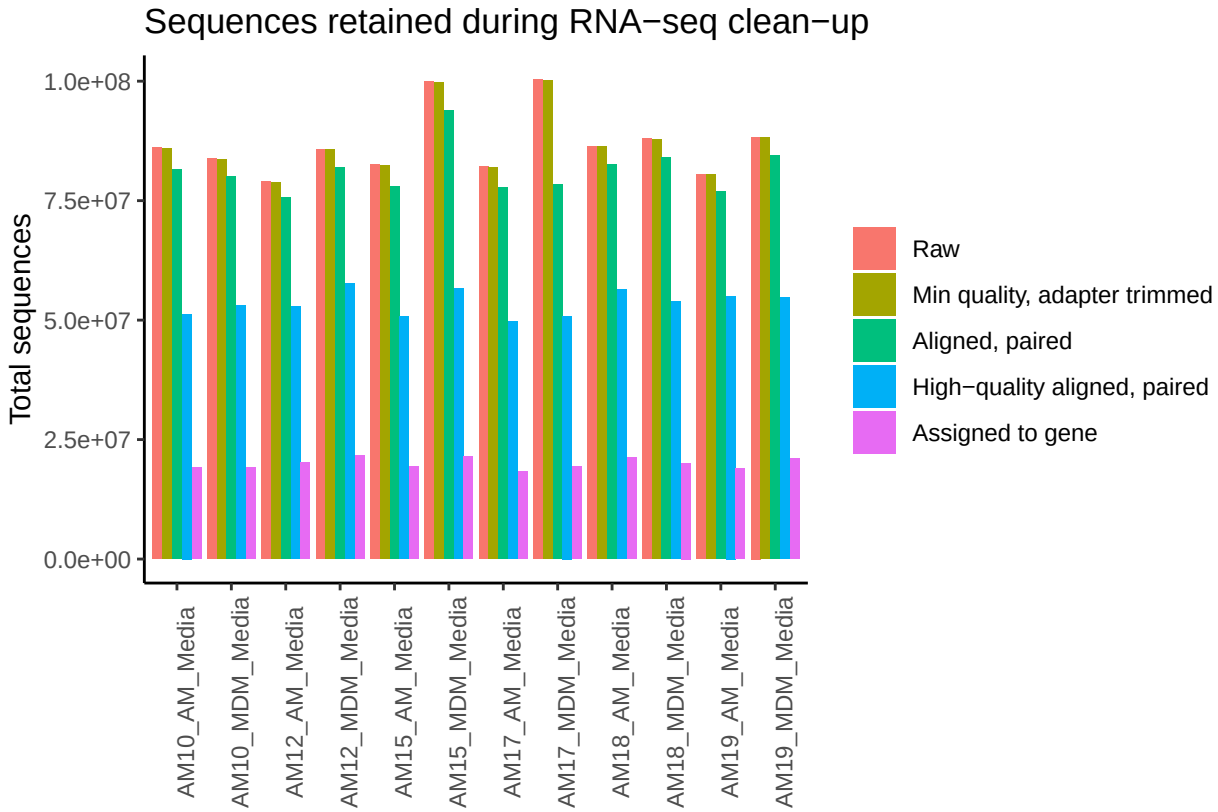
```
#Save the plot to the environment
plot1 <-
#### Data manipulation ####
  raw_data %>%
  #Select variables of interest
  select(sampID, raw, trim,
         both.align.paired,
         both.align.paired_filter,
         Assigned_paired) %>%
  #Filter to Media samples only
  filter(grepl("Media",sampID)) %>%
  #Convert data to long format
  pivot_longer(-sampID, names_to = "group", values_to = "sequences") %>%
  #Convert group into a factor and force the level order
  mutate(group = factor(group, levels=c("raw", "trim", "both.align.paired",
                                        "both.align.paired_filter",
                                        "Assigned_paired"))) %>%

#### Basic plot  ####
  #Initiate ggplot of sequences in each sample,
  ggplot(aes(x=sampID,y=sequences,
             #Fill bar color by the variable group
             #Also tell which variable to group bars by
             fill=group)) +
  #Create a bar plot, grouping all groups from each sample together
  geom_bar(stat="identity", position=position_dodge()) +

#### Customization ####
  #Add a theme to change the overall look of the plot
  theme_classic() +
  #Rotate x-axis labels so we can read them
  theme(axis.text.x = element_text(angle = 90)) +
  #Re-label the legend
  scale_fill_discrete(name="", labels=c("Raw",
                                        "Min quality, adapter trimmed",
                                        "Aligned, paired",
                                        "High-quality aligned, paired",
                                        "Assigned to gene")) +
  #Re-label the axes
  labs(x="",y="Total sequences") +
  #add a title
  ggtitle("Sequences retained during RNA-seq clean-up")
```

Now we can view the plot

```
plot1
```



Save a ggplot

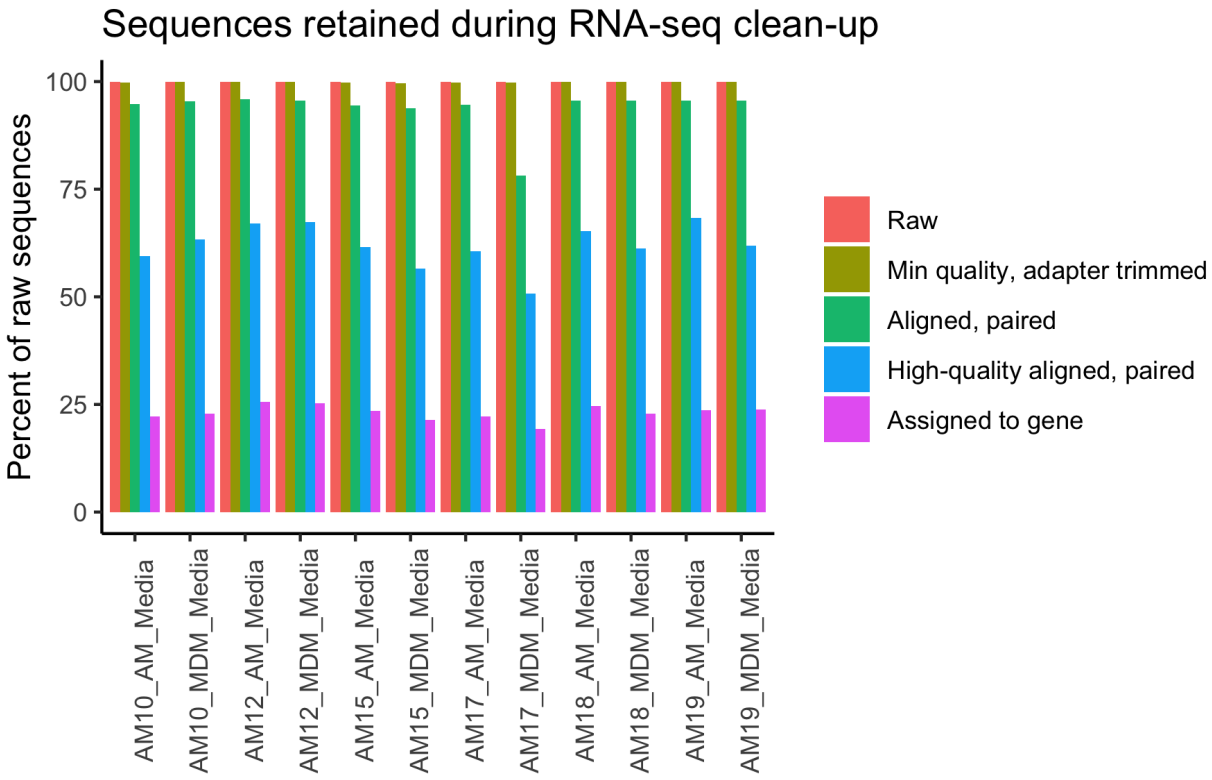
`ggsave` provides a quick way to save plots in a variety of formats (see `?ggsave` for all options). This allows you to reproducibility create figures from a script and can be very helpful when tweaking a figure for publication.

```
ggsave(filename = "images/total.seqs.pdf",
       plot = plot1,
       height=4, width=6)
```

Exercises: Part 2

In these exercises, we challenge you to expand your knowledge by applying new functions from the tidyverse. You'll likely need to use the help pages (and any Googling you want).

1. Rename the `raw` and `trim` columns to more descriptive names using dplyr's `rename` function.
2. Arrange the data frame from most to least raw sequences using dplyr's `arrange` function. *Hint* the default behavior is to arrange in ascending order
3. Further explore the final ggplot function by deleting one or more lines of the customization. Run the plot to see want changes.
4. Challenge: Instead of plotting total sequences, plot values as a percentage of raw sequences in each sample. *Hint* You'll need to use `mutate` and `group_by`. The goal plot looks like



Additional resources

Groups

- Rladies Seattle Not just for ladies! A pro-actively inclusive R community with both in-person and online workshops, hangouts, etc.
- R code club Dr. Pat Schloss is opening his lab's coding club to remote participation.
- Seattle useR Group

Online

- R cheatsheets also available in RStudio under Help > Cheatsheets
- The Carpentries
- Introduction to dplyr
- dplyr tutorial
- dplyr video tutorial
- More functions in dplyr and tidyr
- ggplot tutorial 1
- ggplot tutorial 2
- ggplot tutorial 3
