Course review, and some useful links

Review

This covers most of the material we have gone over through out this course

Base R

```
In R, data is stored using variables. The assignment operator ‘<-‘ and the equals operator ‘=‘ can be us
```

```
x=5
#or
x <- 5
```

Data types

There are four main data types in R: - `numeric`: any number based data, integers, decimals. EX. `5`, or `4.5` - `character`: any textual data. Must be enclosed in `""` or `''`. EX. `"Hello"` - `factor`: a method for representing character data in a memory efficient way. A mapping between unique characters and numbers is created, and only mapping + numerical representation end up being stored - `logical`: Also known as `bools`, takes on two values - `TRUE` or `FALSE`. Generated by logical operators and used to control flow of code

Data Structures

Data structures are containers for multiple variables. Commonly used data strictures in R - Vector: a sequence of values; must be a single type. Attempting to create a vector with mixed data types leads to type coercion

```
#ex
example_vector <- c(1, 2, 3, 4)
```

- List: a sequence of values or other objects. lists can store individual variables, vectors, other lists, or potentially any other data type you may encounter in R.

```
example_list <- list(x=c(1, 2,3, 5),
                     y='Hello')
```

- `data.frame`: a data structure for holding rectangular data made of rows and columns. Under the hood, can be considered a list of columns

```
data.frame(x=c('a', 'b', 'c'),
          y=1, 2, 3)
```

```
##   x y X2 X3
## 1 a 1  2  3
## 2 b 1  2  3
## 3 c 1  2  3
```

Indexing

- most major data structures provide a method for accessing specific elements stored inside them. This is called indexing
- generally each element in a data structure can be accessed by its numeric position within the data structure. See examples below for syntax

```
## vector
x <- c(1,2, 3)
x[2]
```

```
## [1] 2
```

```
# list
example_list <- list(x=c(1, 2,3, 5),
                     y='Hello')
example_list[[1]]
```

```
## [1] 1 2 3 5
```

```
## Data frames
## df[<rowindex>, <columnindex>]
df <- data.frame(x=c('a', 'b', 'c'),
          y=1, 2, 3)
df[2, 3]
```

```
## [1] 2
```

Functions

Functions are re-usable pieces of code that are designed for performing a specific task. Functions can take an input, apply some code to it, and pass back an output. EX.

```
get_length <- function(example_input){
  print(example_input)
  example_output <- length(example_input)
  return(example_output)
}
```

Variables are passed from outside the function inside via the input arguments and can be mapped explicitly using the syntax below. The `return` function passes information from inside the function back to the environment that called it.

```r
d <- c(1, 2,3)
get_length(example_input = d)
```

```
## [1] 1 2 3
```

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

For a more comprehensive overview of base R, see the file `Intro-to-R/intro_to_R_slides.html` in the course folder

Logical operations

We use logical operations to evaluate whether data meets certain conditions. Common logical operators are

- `<`, `>`: compares two numerical items. if expression is logical statement, returns true. EX

```r
5>3
```

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

```r
1>5
```

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

- `==`: equality operator: check whether two variables are exactly the same. EX:

```r
5==5
```

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

```r
5==4
```

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

- `!`: the not operator. Invert an existing logical variable

```r
!TRUE
```

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

Logical operators can be use to generate logical vectors, which can be used to index and filter data

```r
x <- c(1, 2, 3, 4, 5)
x[x>2]
```

```
## [1] 3 4 5
```

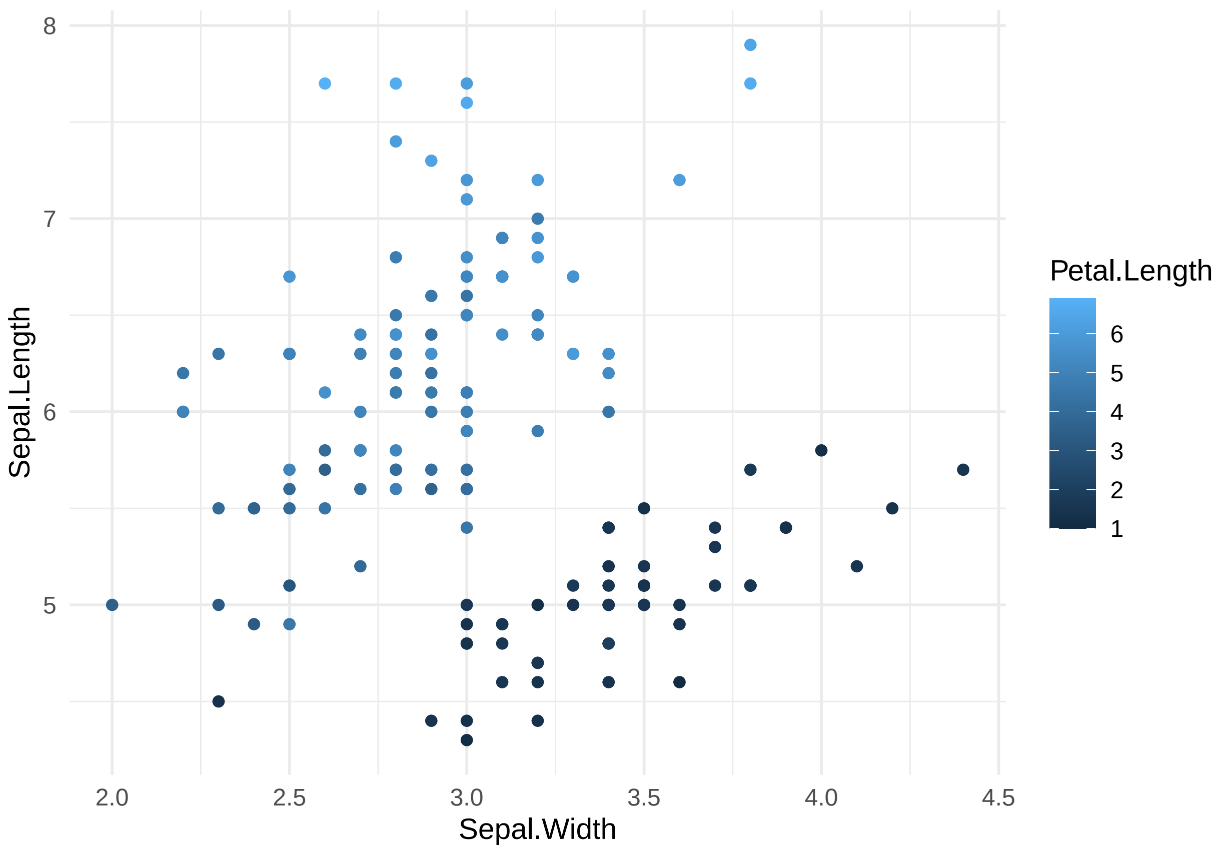
For a more comprehensive overview of logical operations, see the file `Intro-to-R/conditional_programming_slides.html` in the course folder.

Plotting with ggplot

Overview

The `ggplot` library provides a plotting API that allows you to use the same syntax for a wide range of plots

```
ggplot(iris) +
  geom_point(aes(x=Sepal.Width, y=Sepal.Length, color = Petal.Length)) +
  theme_minimal()
```



A plot made with `ggplot` has three key layers

- base layer (`ggplot(...)`): this function initializes a blank plot. both data and aesthetic mappings can be supplied through the `ggplot` function, but are set as the defaults for the rest of the plot. Input data must be a data.frame
- geom layer (`geom_...(...)`): draws a specific visualization, ie boxplot, scatter plot etc. Each distinct `geom_...` function has its own unique set of aesthetic parameters that can either be mapped with the `aes` function to columns in the input data, or set manually. Multiple geom layers can be added together to make complicated plots. Multiple dataset can be included and passed directly to geom layers instead of the base `ggplot()` layer
- theme layer (`theme_...(...)`): the `theme` functions exposes a variety of graphical parameters like axis labels or legend orientation. Several preset themes like `theme_minmal()` have parameters set to specific settings for easy use.

In order to save plots, use the `ggsave` function

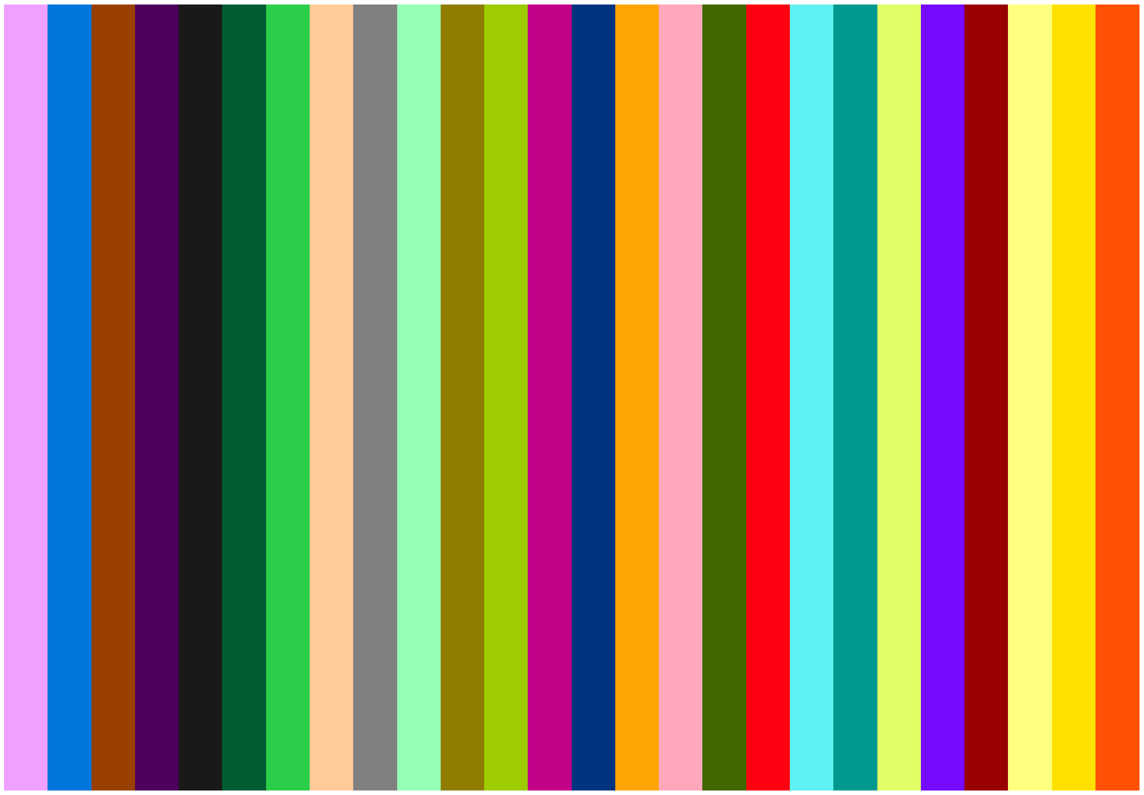
```r
plot <- ggplot(iris) +
  geom_point(aes(x=Sepal.Width, y=Sepal.Length, color = Petal.Length)) +
  theme_minimal()

ggsave(filename = 'example.png', plot = plot, bg = 'white')
```

```
## Saving 6.5 x 4.5 in image
```

Colors in R

A diverse range of colors are available to you in R. Packages like `pals` and `RColorBrewer` provide multiple color palettes. Most data is either categorical or numeric; categorical data is composed of distinct units that are completely separate from each other. They are best represented by divergent color palettes

```r
plot_pal(pals::alphabet(26))
```



Continuous data is best represented by a gradient of three contrasting colors

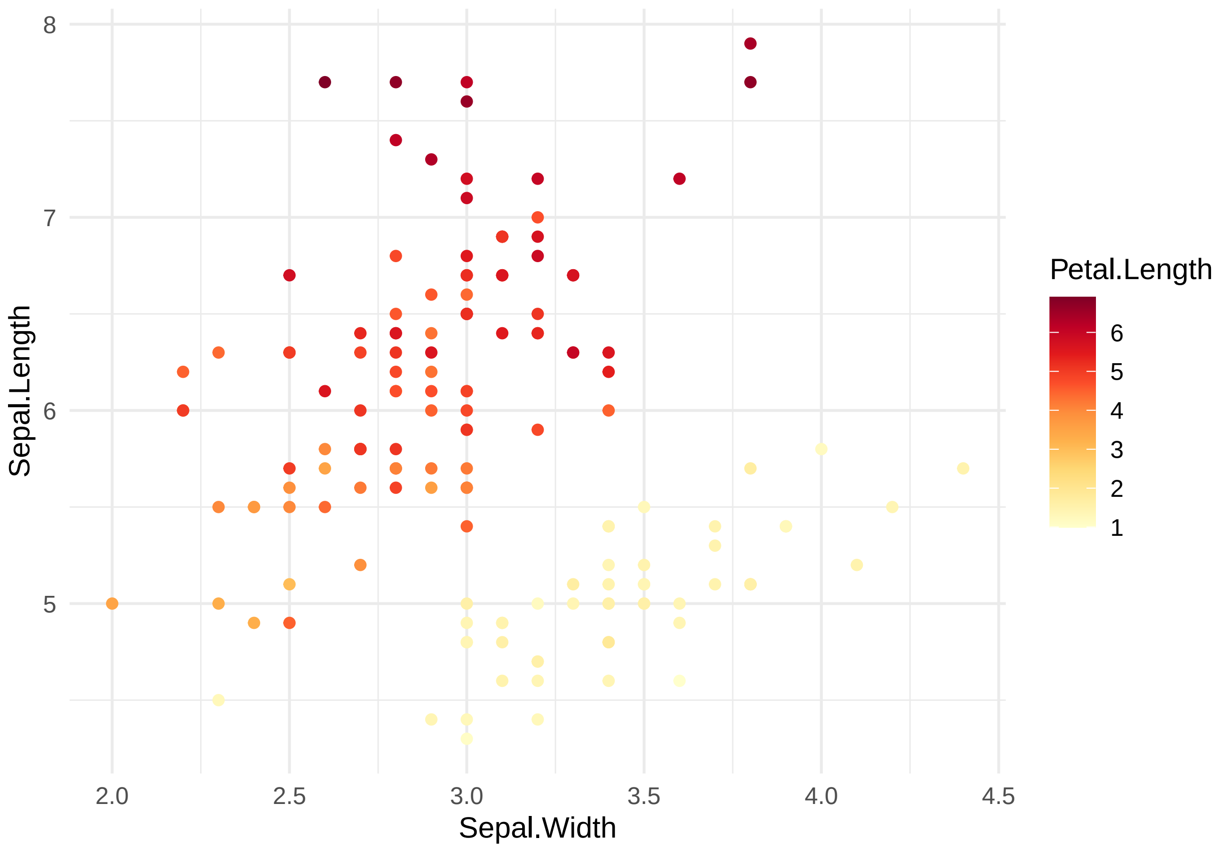
```r
plot_pal(pals::viridis(12))
```

Modifying aesthetics

Generally, any aesthetic parameter that can be set within the `aes`(ie color, fill, size etc) function can be set with some `scale...` function

```
ggplot(iris) +
  geom_point(aes(x=Sepal.Width, y=Sepal.Length, color = Petal.Length)) +
  scale_color_gradientn(colours = RColorBrewer::brewer.pal(12, 'YlOrRd'))+
  theme_minimal()
```

```
## Warning in RColorBrewer::brewer.pal(12, "YlOrRd"): n too large, allowed maximum for palette YlOrRd i
## Returning the palette you asked for with that many colors
```



For a more information about plotting with `ggplot`, see the slides in the `Plotting-with-ggplot/` folder.

Manipulating data with the `tidyverse`

The Tidyverse is a collection of packages that provide fast, easy to use functions for manipulating tabular data in R. To load them all, use

```
library(tidyverse)
```

`readr`

The `readr` library provides funtions for reading/writing data from/to the disk

- reading

```
sc_data <- read_tsv('../src/pbmc_cell_expression.tsv.gz')
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   barcode = col_character(),
##   cell_type = col_character(),
##   cell_type_relabeled = col_character(),
##   patient_id = col_character(),
##   PPBP = col_double(),
##   LYZ = col_double(),
##   S100A9 = col_double(),
##   IGLL5 = col_double(),
##   GNLY = col_double(),
##   FTL = col_double(),
##   PF4 = col_double(),
##   FTH1 = col_double(),
##   GNG11 = col_double(),
##   S100A8 = col_double()
## )
```

```
head(sc_data)
```

```
## # A tibble: 6 x 14
##   barcode   cell_type cell_type_relab~ patient_id  PPBP   LYZ S100A9 IGLL5  GNLY
##   <chr>     <chr>     <chr>            <chr>      <dbl> <dbl>  <dbl> <dbl> <dbl>
## 1 AAACATAC~ Memory C~ Memory CD4 T     patient_1   0     1.64   0        0  0
## 2 AAACATTG~ B         FCGR3A+ Mono     patient_3   0     1.96   0        0  0
## 3 AAACATTG~ Memory C~ Naive CD4 T      patient_2   0     2.00   0        0  1.43
## 4 AAACCGTG~ CD14+ Mo~ Memory CD4 T     patient_3   1.57  4.52   3.84     0  0
## 5 AAACCGTG~ NK        Naive CD4 T      patient_2   0     0      0        0  3.45
## 6 AAACGCAC~ Memory C~ Memory CD4 T     patient_3   0     1.73   0        0  0
## # ... with 5 more variables: FTL <dbl>, PF4 <dbl>, FTH1 <dbl>, GNG11 <dbl>,
## #   S100A8 <dbl>
```

- writing

```
write_csv(sc_data, 'example.csv')
```

dplyr

The `dplyr` library provides multiple functions for manipulating dataframe:

- `select`: select columns from a data.frame

```
iris %>% select(Sepal.Width, Sepal.Length) %>% head
```

```
##   Sepal.Width Sepal.Length
## 1         3.5          5.1
## 2         3.0          4.9
## 3         3.2          4.7
## 4         3.1          4.6
## 5         3.6          5.0
## 6         3.9          5.4
```

- `mutate`: add or change a column

```
iris %>% mutate(is_long_sepal = Sepal.Width >5) %>% head
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species is_long_sepal
## 1          5.1         3.5          1.4         0.2  setosa         FALSE
## 2          4.9         3.0          1.4         0.2  setosa         FALSE
## 3          4.7         3.2          1.3         0.2  setosa         FALSE
## 4          4.6         3.1          1.5         0.2  setosa         FALSE
## 5          5.0         3.6          1.4         0.2  setosa         FALSE
## 6          5.4         3.9          1.7         0.4  setosa         FALSE
```

- `rename`: change the name of a columns

```
iris %>% rename(length_sepal = Sepal.Length) %>% head
```

```
##   length_sepal Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

- `arrange`: sort data by values in 1 or more columns

```
iris %>% arrange(Species, Sepal.Length) %>% head
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          4.3         3.0          1.1         0.1  setosa
## 2          4.4         2.9          1.4         0.2  setosa
## 3          4.4         3.0          1.3         0.2  setosa
## 4          4.4         3.2          1.3         0.2  setosa
## 5          4.5         2.3          1.3         0.3  setosa
## 6          4.6         3.1          1.5         0.2  setosa
```

- `filter`: subset data based on logical operations

```
iris %>% filter(Sepal.Length >5 ) %>% head
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          5.4         3.9          1.7         0.4  setosa
## 3          5.4         3.7          1.5         0.2  setosa
## 4          5.8         4.0          1.2         0.2  setosa
## 5          5.7         4.4          1.5         0.4  setosa
## 6          5.4         3.9          1.3         0.4  setosa
```

- `group_by` and `summarise`: group rows by unique values across multiple columns, and then perform a operations on each group.

```
iris %>%
  group_by(Species) %>%
  summarise(avg_petal_length = mean(Petal.Length))
```

```
## # A tibble: 3 x 2
##   Species    avg_petal_length
##   <fct>                 <dbl>
## 1 setosa                 1.46
## 2 versicolor             4.26
## 3 virginica              5.55
```

tidyr

There's really only one function we covered from here - `pivot_longer`: convert data from wide to long format

```
iris %>%
  pivot_longer(cols = -Species, names_to = 'measurement_type', values_to = 'measuerment_values')  %>% he
```

```
## # A tibble: 6 x 3
##   Species measurement_type measuerment_values
##   <fct>   <chr>                         <dbl>
## 1 setosa  Sepal.Length                    5.1
## 2 setosa  Sepal.Width                     3.5
## 3 setosa  Petal.Length                    1.4
## 4 setosa  Petal.Width                     0.2
## 5 setosa  Sepal.Length                    4.9
## 6 setosa  Sepal.Width                     3
```

For more information about the `tidyverse` see the slides in `Using-the-tidyverse`

Libraries used througout the course

ComplexHeatmap

- Non-ggplot based package for plotting Heatmaps.
- Heatmaps are useful visualizations for showing subpopulations in clustered data
- Has extensive documentation and is extremely flexible.

ComplexUpset

- Non-ggplot based package for plotting upset plots
- upset plots are useful for visualizing interactions between sets

DT

- Create interactive data tables in Rmarkdown and Shiny

RColorBrewer

- create diverse color palettes for plots. Can generate both discrete and continuous palettes

circlize

- Create circos plots and circular color scales

dplyr

- an *excellent* toolbox for cleaning and manipulating data

flextable

- create precisely formatted tables when using Rmarkdown to generate `.docx` files

gapminder

- load the `gapminder` dataset as a `data.frame`, and contains information about population, life expectancy and GDP.
- Use data set for testing out plots

ggalluvial

- Generate Alluvial plots using ggplot syntax
- alluvial plots are useful for visualizing subsets in categorical data, or changes in composition of categorical data over time

gganimate

- Turn (almost) any ggplot into an animated figure.

gghalves

- Plot half geoms(violin/boxplot/points)
- useful for showing changes between two similar conditions

ggiraph

- Uses ggplot syntax to add interactivity to plots by adding tooltip(show info when hovering mouse over position)

ggpattern

- Add shading patterns to ggplots

ggplot2

- The best plotting library in R. uses the Grammar of Graphic(gg) to define a concise syntax for generating plots
- `ggplots` are composed of several key components

  - `ggplot(data)` : create a new blank plot
  - `geom_...` : add some sort of visualization(points, boxplots, etc), and
  - `aes()` : generate and aesthetic mapping between columns in `data` and visual aesthetics in `geom_...`
  - save plots using `ggplot::ggsave`

ggplotify

- Convert any plot generated in R into a `ggplot` object.
- This allows plots generated from non-ggplot based libraries to be merged with other plots through `patchwork`, or saved using `ggplot::ggsave`

ggpubr

- an alternative plotting package that provides simpler functions for generating plots, and provides functions for for may

ggrepel

- generate text and labels on `ggplots` that *repel* from each other, so they don't crowd each other, or other elements in the plot

ggridges

- create ridge plots in `ggplot`.
- ridge plots are useful for visualizing continuous distributions across many variables

ggsci

- provides alternative color palettes that are inspired by plots from different scientific journals

ggspatial

- an extension to ggplot for plotting map data.
- uses spatial data defined by the `sf` package

ggthemes

- a collection of themes and color palettes inspired by popular websites

ggtree

- an extension to `ggplot` for visualizing trees and dendrograms

kableExtra - An extension to `Rmarkdown` that allows you to better format tables

knitr

- the core library behind `Rmarkdown`
- use knitr to control `Rmarkdown` and create high-quality reports and documents with embedded code

maps - provides spatial info for common maps(world, USA, etc)

pals

- a comprehensive set of color palettes for R
- in my opinion, the best package for generating color palettes

patchwork - a useful library for assembling multiple `ggplots` into a single figure - combine plots with intuitive operators like `+`, `/` and , `|`, as well as specify custom layouts

plotly

- a feature-rich library for making interactive visualizations that can be embedding in `Rmarkdown` reports or `shiny` apps

redoc

- seamlessly(ish) move between Rmarkdown <==> Microsoft Word

sf

- a package for working with geospatial data in R

shiny

- the core package for interactive visualizations in R.
- Build fast and dynamic webapps that can be hosted locally or on a webserver

tidyverse

- an *amazing* collection of packages for data science in R.
  - tibble: a new data type analogous to a `data.frame`, but with more consistent indexing, and no type coercion
  - dplyr: a series of functions for subsetting, filtering, changing, and summarizing dataframes/tibbles
  - tidyr: functions for tidying data(more on that later)
  - stringr: library for efficiently working with strings
  - readr: efficiently read/write data
  - forcats: library for efficiently working with factors
  - purrr: adds common programming methods from other languages into
  - using `library(tidyverse)` makes it easy to load these all at once; but you can load them individually if you choose

Useful links

Rstudio cheatsheets

Quick references for common packages. See more cheatsheets here

- base R : tips and tricks for base R
- dplyr : manipulating and tidying data
- ggplot : core plotting library
- shiny : interactive visualizations
- Rmarkdown : creating notebooks and reports