UA Summer Workshop: 31 May 2016

Nicholas M. Caruso

Intro to R language

Here is some basic code to get accustomed to the language. You can easily perform calculations, use many of the built in functions for many mathematical operations. There are too many functions to go over today, but we will cover more as we move along.

```
1 + 4
7/5
log(10) # calculate the natural log of a number/vector
log10(5) # calculate the log base 10 of a number/vector
rnorm(n=10, mean=0, sd=1) # vector of n random numbers with a mean and standard deviation
seq(from=0, to=10, by=1) # vector with min and max, with known increments or length
rep(1, times=10) # repeat a number or vector
rep(c(1,2), each=5); rep(c(1,2), times=5) #each and times specify repeats differently
x <- c(1,2,3); # store an object into a name
length(x) # length of object
mean(x) # mean of object
max(x) #maximum of object
```

Dataframes

Dataframes are similar to excel spread sheets, they are arranged by row and column, here are some basic functions to inspect and manipulate dataframes (lots more on this later!)

```
iris #included in base R
str(iris) #look at structure
dim(iris) #dimensions
head(iris) #first 6 rows
tail(iris) #last 6 rows
colnames(iris) #column names
iris$Sepal.Length # Sepal.Length column
iris[['Sepal.Length']] #Sepal.Length column again
iris[1,] #first row, all columns
iris[,1] #first column, all rows
iris[1,1] #first row, first column
```

dplyr: a package for manipulating dataframes

This overview will include functions for filtering, selecting, summarising, mutating, and joining datasets. First some new functions in *dplyr* that allow us to inspect dataframes.

```
# install.packages('dplyr') #install dplyr package, only need to do this once
library(dplyr) #use library function to load the package

# convert dataframe to tbl, much easier to examine
iris.tbl <- tbl_df(iris)
iris; iris.tbl
glimpse(iris.tbl) # similar to str
View(iris.tbl) # view the whole dataframe
arrange(iris.tbl, Sepal.Width) # arrange dataframe by column(s)
```

dplyr package makes use of "piping" %>% to pass functions to dataframes, creating more readable code. Plus there are lots of great functions in dplyr to help with data manipulation!

Filtering Datasets

Use == for exact matches, %in% for multiple matches and != for exclusions (does not equal). Can also use greater than/less than for numeric columns (>= for greater than and equal to). Use & to string multiple filtering arguments together

```
iris.tbl %>%
  filter(Species=='setosa')

iris.tbl %>%
  filter(Species %in% c('setosa','versicolor'))

iris.tbl %>%
  filter(Species != 'setosa')

iris.tbl %>%
  filter(!Species %in% c('versicolor','virginica'))

iris.tbl %>%
  filter(Sepal.Width > 3 & Species=='setosa')
```

Select Columns in Dataframes

Can specify which columns to select or remove (denoted with '-'). Can have multiple arguments.

```
iris.tbl %>%
  select(Species)

iris.tbl %>%
  select(Sepal.Length, Species)

iris.tbl %>%
  select(-Species)
```

Summarise Dataframes

The summarise functions are useful for making summary stats (e.g., mean, sd, etc) of your data. These functions take a vector of values and return a single value. Can be used in combination with group_by() function to examine different levels of a factor.

```
iris.tbl %>%
  summarise(mn.sepalWidth=mean(Sepal.Width),
            sd.sepalWidth=sd(Sepal.Width))

iris.tbl %>%
  group_by(Species) %>%
  summarise(mn.sepalWidth=mean(Sepal.Width),
            sd.sepalWidth=sd(Sepal.Width))

iris.summary <- iris.tbl %>%
  group_by(Species) %>%
  summarise_each(funs(mean, sd, n(), min, max, var, median))

View(iris.summary)
```

Mutate Functions

The mutate family of functions are useful for making new columns from your existing dataset (e.g., log-transforming a column). These functions take a vector of values and return a vector of values that is the same length. Can be applied over the whole column or used with group_by() similarly to summarise functions.

```
iris.tbl %>%
  mutate(log.sepalWidth=log(Sepal.Width))

View(iris.tbl %>%
  group_by(Species) %>%
  mutate(mn.sepalWidth=mean(Sepal.Width),
         sd.sepalWidth=sd(Sepal.Width),
         sepalWidth.stnd=(Sepal.Width-mn.sepalWidth)/sd.sepalWidth))

# Some Useful mutate functions that I typically use
## Lag = previous value
## Lead = next value
## between = Logical, is the value between any two values
## cumsum = cumulative sum

iris.tbl %>%
  select(Sepal.Width) %>%
  mutate(lag(Sepal.Width),
         lead(Sepal.Width),
         between(Sepal.Width, 3, 4),
         cumsum(Sepal.Width))
```

Join Functions

The join functions combines rows of dataframes based on identifiers (e.g., date, time, individual). We'll create our own dataframes to demonstrate these functions.

```r
dat1 <- data_frame(x1=c(25,14,68,79,64,139,49,119,111),
                   x2=rep(letters[1:3], each=3),
                   x3=c(1,14,22,2,0,20,2,13,22))


dat2 <- data_frame(x2=rep(letters[c(1,2,4)], each=2),
                    y1=c(1:6),
                    y2=rep(c(22,2,0), times=2))

dat1; dat2

(new.dat <- left_join(x=dat1, y=dat2, by='x2')) #join from b to a
right_join(dat1, dat2, by='x2') # joing from a to b

full_join(dat1, dat2, by='x2') # keep all
inner_join(dat1, dat2, by='x2') #only keep matching in both
semi_join(dat1, dat2, by='x2') # keep rows in a that match b
anti_join(dat1, dat2, by='x2') # only keep a that don't match b

## Can join by multiple matching columns

dat1 <- dat1 %>%
  rename(y2=x3)

inner_join(dat1, dat2, by=c('x2','y2'))
```

tidyr: a package for changing the shape of datasets

This overview will include functions for changing the shape (layout) of datasets. The two functions I typically use are gather(), which gathers columns into rows (wide format to long format) and spread(), which spreads rows into columns (long format to wide).

```
# install.packages('tidyr')
# install.packages('lubridate')
library(tidyr)
library(lubridate)

dat <- data_frame(ind=c(1,2,3), trial1=c(20,30,40), trial2=c(60,70,80),
                  date=dmy(c('26-05-2016','27-05-2016','28-05-2016')))

dat.g <- dat %>%
  gather(key=trial, value=response, trial1:trial2)

dat.g %>%
  spread(key=trial, value=response)

dat %>%
  separate(date, c('year','month','day'), sep='-') %>%
  mutate_each_(funs(as.numeric), c('year','month','day')) %>%
  gather(trial, response, trial1:trial2)

## Or if we want Julian day

dat %>%
  mutate(julian.day=yday(date)) %>%
  gather(trial, response, trial1:trial2)
```

There are a lot of useful functions in the *lubridate* package for working with date objects.

ggplot2: a package for data visualization

ggplot2 is a great alternative to the already flexible plotting included with base R. However, the advantage of ggplot is that it every graphic made is based on three components: the data, the coordinate system (e.g., x and y), and a set of visuals to display the data (geoms). First we build the graph with some data and the aesthetic properties (aes) of the graphic (e.g., x, y, groupings, shape of points, color, linetypes). Next we'll add some geoms to represent the data

ggplot2 basics

```
# install.packages('ggplot2')
library(ggplot2)

p <- ggplot(data=iris, aes(x=Sepal.Width, y=Sepal.Length))

p + geom_point()
```

We can include additional aesthetics to deliniate species in the graphic. For comparison here's how you would plot a similar graphic in base R.

```
ggplot(data=iris, aes(x=Sepal.Width, y=Sepal.Length, shape=Species, color=Species)) +
  geom_point()

plot(Sepal.Length~Sepal.Width, data=iris, pch=as.numeric(iris$Species),
     col=as.numeric(iris$Species))
```

Its pretty obvious how much better the default ggplots look compared to base R. I'll clean up the plot a little bit more using ggplot to demonstrate what typically goes into making a more polished graphic. Multiple visual representations (geoms, themes, labels, etc) need to be separate by a '+'. For this graphic, we will change the x and y labels, apply a simple theme (there are a lot of ggthemes, or you can create your own), add linear model predictions (with standard errors) for each species, and move the legend to inside the plot to use the full plotting space and make the species names italics. Note that since we set up the aestics for species in the beginning of the plot, we don't have to do it again for the linear model predictions, but we could add aesthics within any geom (e.g., change line width, lwd) If we want to use one value for all levels, we would describe that aesthetic outside of the aes() function (e.g., alpha).

Scatterplot

```
ggplot(data=iris, aes(x=Sepal.Width, y=Sepal.Length, shape=Species, color=Species)) +
  geom_point() +
  labs(x='Sepal Width',
       y='Sepal Length') +
  geom_smooth(method='lm', formula=y~x, se=TRUE) +
  theme_bw() + # classic dark on light theme
  theme(legend.position=c(0.1, 0.85), # c(x, y) assume both are [0,1]
        legend.background=element_rect('transparent'),
        legend.text=element_text(face='italic')) +
  scale_color_manual(values=c('red','blue','black'))
```

Typical geoms

There are a lot more geoms to produce many different types of plots, here are the ones that I typically use
geom_density() distribution of variables
geom_histogram() histogram
geom_ribbon() typically used for showing confidence interval bands
geom_jitter() move points small increments if they overlap (e.g., binomial data)
geom_point() plot two continuous variables with a scatterplot
geom_smooth() add lm, glm, and loess smoothed fits to data
geom_bar() barplot
geom_boxplot() boxplot
geom_violin() violin plot
geom_desnity2d() 2 dimensional density contours
geom_errorbar() geom_errorbarh() error bars for vertical and horiztonal intervals respectively
geom_polygon() typically used to make background maps (use map_data() to retrieve reference maps)
geom_raster() plot rasters, must be transformed to a dataframe first

Examples

```
# Density plot
ggplot(iris, aes(x=Sepal.Width, fill=Species)) +
  geom_density(alpha=0.7)

# Violin plot with jittered points
ggplot(iris, aes(x=Species, y=Sepal.Width)) +
  geom_violin(color='gray') +
  geom_jitter(width=0.25, height=0.001)

# Boxplot
ggplot(iris, aes(x=Species, y=Sepal.Width)) +
  geom_boxplot()

# Barplot - Stacked
ggplot(mpg, aes(x=class, fill=drv)) +
  geom_bar()

# Barplot, side by side
ggplot(mpg, aes(x=class, fill=drv)) +
  geom_bar(position='dodge')

# Barplot, filled
ggplot(mpg, aes(class, fill=drv)) +
  geom_bar(position='fill')

# 2d contour
ggplot(iris, aes(Sepal.Width, Sepal.Length)) +
  geom_density2d() +
  geom_point()

# maps
# install.packages('maps')
library(maps)
us.map <- map_data('state')
```

```
bama <- data_frame(long=-87.5692, lat=33.2098)
# install.packages('emoGG')
# devtools::install_github('dill/emoGG')
library(emoGG)

emoji_search("elephant")

ggplot(bama)+
  geom_polygon(data=us.map, aes(x=long, y=lat, group=group), fill='transparent',
               color='black') +
  geom_emoji(emoji='1f418', aes(x=long, y=lat))

# raster
# install.packages('raster')
library(raster)

bio.clim <- getData('worldclim', var='bio', res=10)

bio1.df <- as.data.frame(bio.clim[[1]], xy=TRUE)
bio1.df <- bio1.df %>%
  na.omit()

ggplot(bio1.df, aes(x=x, y=y)) +
  geom_raster(aes(fill=bio1)) +
  scale_fill_gradient2(low='blue', high='red')
```

Facets

A nice feature of ggplots is facets, which divide a plot into subplots based on categorical (discrete) variables. There are two ways to facet plots facet_wrap() and facet_grid(). Wrap will only plot those subplots that have data, while grid will plot all combinations regardless of data, this is demonstrated below. Use the labeller argument to adjust how the facets are labelled.

```
mpg

ggplot(mpg, aes(x=cty, y=hwy)) +
  geom_point() +
  facet_grid(year~.)

ggplot(mpg, aes(x=cty, y=hwy)) +
  geom_point() +
  facet_grid(.~cyl)

ggplot(mpg, aes(x=cty, y=hwy)) +
  geom_point() +
  facet_grid(year~cyl, labeller=label_both)

ggplot(mpg, aes(x=cty, y=hwy)) +
  geom_point() +
  facet_wrap(year~cyl)
```

Multiple panels

If you have multiple plots that you want to panel together in *ggplot2*, you will have to use the *gridExtra* package. The default is to give each plot equal sizes but you can adjust the layout to make them different sizes.

```
# install.packages('gridExtra')
library(gridExtra)

p1 <- ggplot(mpg, aes(x=cty, y=hwy)) +
  geom_point()

p2 <- ggplot(iris, aes(x=Species, y=Sepal.Width)) +
  geom_violin(color='gray') +
  geom_jitter(width=0.25, height=0.001)

grid.arrange(p1, p2, ncol=2)

p3 <- ggplot(bama)+
  geom_polygon(data=us.map, aes(x=long, y=lat, group=group), fill='transparent',
               color='black') +
  geom_emoji(emoji='1f418', aes(x=long, y=lat))


grid.arrange(p1, p2, p3, layout_matrix = rbind(c(1,2),
                                               c(3,3)))
```

Alternatively you can use the *cowplot* package which automatically changes the theme (you can use any theme though) and includes a nice feature for adding plot identifiers (e.g., A, B, C, etc).

```
# install.packages('cowplot')
library(cowplot)

ggdraw() +
  draw_plot(plot=p1, x=0, y=0.5, width=0.5, height=0.5) +
  draw_plot(p2, 0.5, 0.5, 0.5, 0.5) +
  draw_plot(p3, 0, 0, 1, 0.5) +
  draw_plot_label(label=c('A','B','C'), x=c(0,0.5,0), y=c(1,1,0.5),
                            size=15)
```