Importing & Exporting Data in R

- R refers to datasets as dataframes.
- A data frame is a matrix-like structure, where the columns can be of different types.
- You can also think of a dataframe as a list. Each column is an element of the list and each element has the same length.
- To get/replace elements of a dataframe use either `[ ]` or `$`. The `[ ]` are used to access rows and columns and the `$` is used to get entire columns.

Importing & Exporting Data in R

```
# state.x77 is a built-in R dataset stored as a matrix
# Type data(), to see a list of built-in datasets

data <- data.frame(state.x77) # First, convert to a dataframe

head(data) # Print the first few rows of a dataset/matrix

tail(data) # Print the last few rows of a dataset/matrix

names(data) # Column names

colnames(data); rownames(data) # Column and row names

dim(data) # Dimension of the dataframe

data[,c("Population", "Income")] # Population and Income columns

data$Area # Get the column "Area"

data[1:5,] # Get the first five rows
```

Importing Data from Files

- The function `read.table()` is the easiest way to import data into R .
- The preferred raw data format is either a tab delimited or a comma-separate file (CSV).
- The simplest and recommended way to import Excel files is to do a Save As in Excel and save the file as a tab delimited or CSV file and then import this file in to R .
- Similarly, for SAS files export the file as a tab delimited or CSV file using `proc export`.
- Functions for importing data,

`read.table()`	Reads a file in table format and creates a dataframe
`read.csv()`	same as `read.table()` where `sep=","`
`read_csv()`	from `readr` package
`read.fwf()`	Read a table of fixed width formatted data. Data that is not delimited, need to specify the width of the fields.

read.table()

```
read.table(file, header = FALSE, sep = " ", skip, as.is,
                    stringsAsFactors=TRUE)
```

`file`	The name of the file to import
`header`	Logical, does the first row contain column labels
`sep`	Field separator character `sep=" "` space (default) `sep="\t"` tab-delimited `sep=","` comma-separated
`skip`	Number of lines to skip before reading data
`as.is`	Vector of numeric or character indices which specify which columns should not be converted to factors
`stringsAsFactors`	Logical should character vectors be converted to factors

- By default R converts character string variables to factor variables, use `stringsAsFactors` or `as.is` to change the default.
- There are many more arguments for `read.table` that allow you to adjust for the format of your data.

Example - read.table()

```
data <- read.table("example.csv", header=TRUE, sep=",", skip=7)
str(data) # Gives the structure of data

# Have all character strings not be treated as a factor variable
data <- read.table("example.csv", header=TRUE, sep=",", skip=7,
                    stringsAsFactors=FALSE)
str(data)

# Have character strings in selected columns not be treated as
# a factor
data <- read.table("example.csv", header=TRUE, sep=",", skip=7,
  as.is="dr")
str(data)
```

Export Datasets

```
write.table(x, file = " ", sep = " ", row.names=TRUE,
                    col.names=TRUE)
```

`x`	The object to be saved, either a matrix or dataframe
`file`	File name
`sep`	Field separator
`row.names`	Logical, include row.names
`col.names`	Logical, include col.names

- There is also a wrapper function, `write.csv()`, for creating a CSV file by calling `write.table()` with `sep=","`.

```
# Export R dataframe as a CSV file
write.table(data, "export.example.csv", sep=",", row.names=FALSE)
```

Modifying Datasets

Let's work on an example using `example_data.csv`.

```
data <- read.table('example_data.csv',sep=",", header=TRUE,skip=7);
```

- `sep=","` means variables are comma separated.
- `header=TRUE` means that there are column names in the first row of the dataset.
- Note that 7 lines are skipped before reading data.

For a few quick checks on the properties of your data, type the following commands into `R`, followed by the Enter key:

- `names(data)`: This returns variable names. `R` assigned generic ones since you didn't provide any.
- `dim(data)`: This returns the number of rows and columns.
- `str(data)`: This shows the format for each variable (”numeric” (= continuous), ”integer”, ”string”, etc).
- `describe(data)`: For a quick table of descriptives for each variable. Install package `Hmisc` first.

Modifying Datasets

- If you want all character strings not be treated as a factor variables, use instead

```
data <- read.table('example_data.csv',sep=",", header=TRUE,skip=7,
              stringsAsFactors=FALSE);
```

- Suppose you want to have character strings in selected columns, say `dr`, not be treated as a factor. Then, use

```
data <- read.table('example_data.csv',sep=",", header=TRUE,skip=7,
              as.is="dr");
```

Modifying Datasets

- Let's have the start date to be a `Date` object. To do so, use

```
data <- read.table('example_data.csv',sep=",", header=TRUE,skip=7,
                   as.is="dr");
Visit.Date <- as.Date(data$visit, "%m/%d/%Y");
cbind(data[,-2], Visit.Date);
```

```
   ID     trt PCS MCS    dr Visit.Date
1   1 Placebo  88  71 Dr. A 2008-03-05
2   1 Placebo  67  68 Dr. A 2008-09-07
3   1 Placebo  67  68 Dr. A 2008-09-07
4   2    Drug  51  64 Dr. Y 2008-06-19
5   2    Drug  86  56 Dr. Y 2008-12-22
6   3 Placebo  85  59 Dr. K 2008-04-11
7   3 Placebo  67  64 Dr. K 2008-10-14
8   4    Drug  71  86 Dr. R 2008-10-21
9   4    Drug  82  64 Dr. R 2009-04-25
10  5    Drug  89  52 Dr. V 2008-07-30
11  5    Drug  72  68 Dr. V 2009-02-01
12  6 Placebo  58  78 Dr. O 2008-02-25
13  6 Placebo  75  60 Dr. O 2008-08-29
14  6 Placebo  75  60 Dr. O 2008-08-29
```

Modifying Datasets: read!!!

The function `search()` displays the search path for R objects. When `R` looks for an object it first looks in the global environment then proceeds through the search path looking for the object. The search path lists attached dataframes and loaded libraries. The function `attach()/detach()` attaches/detaches a dataframe to the search path. This means that the column names of the dataframe are searched by `R` when evaluating a variable, so variables in the dataframe can be accessed by simply giving their names. `attach()` is okay if you are just working on one dataset and your purpose is mostly on analysis, but if you are going to have several datasets and lots of variables avoid using `attach()`. If you attach a dataframe and use simple names like `x` and `y`, it is very possible to have very different objects with the same name which can cause problems. `R` prints a warning message if attaching a dataframe causes a duplication of one or more names. Several modeling functions like `lm()` and `glm()` have a data argument so there is no need to attach a dataframe. For functions without a data argument use `with()`. This function evaluates an `R` expression in an environment constructed from the dataframe. If you do use `attach()` call `detach()` when you are finished working with the dataframe to avoid errors. The `scan()` function is very flexible, but as a result is also harder to use then `read.table()`. However, `scan()` can be useful when you need to import odd datasets.

Modifying Datasets

To sort dataframe by row, use order and []. Let's use CO2 is a built-in R dataset on cold tolerance of grass species.

```
attach(CO2);
order(uptake);
uptake[order(uptake)];
# Sort CO2 by uptake, descending
CO2[order(uptake),];
# Sort CO2 by uptake, ascending
CO2[rev(order(uptake)),];
CO2[order(uptake, decreasing=TRUE),];
# Sort CO2 by conc then uptake
CO2[order(conc, uptake),];
# Sort CO2 by conc (ascending) then uptake (descending), only for nume
CO2[order(conc, -uptake),]
# Sort CO2 by Plant, an ordered factor variable
CO2[order(Plant),]
# Sort CO2 by Plant (descending) then uptake (ascending)
# rank() converts Plant to numeric
CO2[order(-rank(Plant), uptake),];
detach(CO2);
```

Modifying Datasets: duplicated obs

- Quite often you will have to deal with duplicate observations. The function `unique()` will return a dataframe with the duplicate rows or columns removed.
- You can use the incomparables argument to specify what values not to compare.
- `duplicated()` returns a logical vector indicating which rows are duplicates.
- Note that `unique()` and `duplicated()` only work for imported dataframes and does not work for dataframes created during an `R` session.
- The `match()` function can be used to select the first entry of a variable.

Modifying Datasets: duplicated obs

```
data <- read.table('example_data.csv',sep=",", header=TRUE,skip=7,as.i
Visit.Date <- as.Date(data$visit, "%m/%d/%Y");
data <-cbind(data[,-2], Visit.Date);

unique(data); # Dataset with duplicate rows removed

   ID     trt PCS MCS    dr Visit.Date
1   1 Placebo  88  71 Dr. A 2008-03-05
2   1 Placebo  67  68 Dr. A 2008-09-07
4   2    Drug  51  64 Dr. Y 2008-06-19
5   2    Drug  86  56 Dr. Y 2008-12-22
6   3 Placebo  85  59 Dr. K 2008-04-11
7   3 Placebo  67  64 Dr. K 2008-10-14
8   4    Drug  71  86 Dr. R 2008-10-21
9   4    Drug  82  64 Dr. R 2009-04-25
10  5    Drug  89  52 Dr. V 2008-07-30
11  5    Drug  72  68 Dr. V 2009-02-01
12  6 Placebo  58  78 Dr. O 2008-02-25
13  6 Placebo  75  60 Dr. O 2008-08-29

data[duplicated(data),]; # Duplicate rows

   ID     trt PCS MCS    dr Visit.Date
3   1 Placebo  67  68 Dr. A 2008-09-07
14  6 Placebo  75  60 Dr. O 2008-08-29
```

Modifying Datasets: duplicated obs

```
data.nodup <- unique(data);
```

- Sort by ID then visit date so that the first entry for each patient is the first visit. Then, find the first visit of each patient.

```
data.nodup.sort<-with(data.nodup, data.nodup[order(ID, Visit.Date),]);
first <- with(data.nodup.sort, match(unique(ID), ID));
data.nodup.sort[first,];
```

```
   ID     trt PCS MCS    dr Visit.Date
1   1 Placebo  88  71 Dr. A 2008-03-05
4   2    Drug  51  64 Dr. Y 2008-06-19
6   3 Placebo  85  59 Dr. K 2008-04-11
8   4    Drug  71  86 Dr. R 2008-10-21
10  5    Drug  89  52 Dr. V 2008-07-30
12  6 Placebo  58  78 Dr. O 2008-02-25
```

Modifying Datasets: merging

- You can use `merge()` to merge two, and only two dataframes.

```
data1 <- data.frame(ID=1:5, x=letters[1:5]);
data2 <- data.frame(ID=1:5, y=letters[6:10]);
merge(data1, data2);
```

```
  ID x y
1  1 a f
2  2 b g
3  3 c h
4  4 d i
5  5 e j
```

Modifying Datasets: merging

- Merge two datasets by an ID variable where ID is not the same for both datasets.

```
data1 <- data.frame(ID=1:5, x=letters[1:5]);
data2 <- data.frame(ID=4:8, y=letters[6:10]);
merge(data1, data2);
```

```
  ID x y
1  4 d f
2  5 e g
```

Modifying Datasets: merging

- Keep all unmatched IDs from both data sets

```
merge(data1, data2, all=TRUE);
```

```
  ID    x    y
1  1    a <NA>
2  2    b <NA>
3  3    c <NA>
4  4    d    f
5  5    e    g
6  6 <NA>    h
7  7 <NA>    i
8  8 <NA>    j
```

Modifying Datasets: merging

- Only keep the unmatched rows from dataset 1

```
merge(data1, data2, all.x=TRUE); # all.y=TRUE for dataset 2
```

```
  ID x    y
1  1 a <NA>
2  2 b <NA>
3  3 c <NA>
4  4 d    f
5  5 e    g
```

Modifying Datasets: merging

- Merge two datasets by an ID variable, where both dataset have the same names

```
data1 <- data.frame(ID=1:5, x=letters[1:5]);
data2 <- data.frame(ID=1:5, x=letters[6:10]);
merge(data1, data2, all=TRUE) # Add rows

   ID x
1   1 a
2   1 f
3   2 b
4   2 g
5   3 c
6   3 h
7   4 d
8   4 i
9   5 e
10  5 j

merge(data1, data2, by="ID",suffixes=c(1, 2));

  ID x1 x2
1  1  a  f
2  2  b  g
3  3  c  h
4  4  d  i
5  5  e  j
```

Modifying Datasets: merging

- Merge two datasets by an ID variable, where the ID variable has a different name

```
data1 <- data.frame(ID1=1:5, x=letters[1:5]);
data2 <- data.frame(ID2=1:5, x=letters[6:10]);
merge(data1, data2, by.x="ID1", by.y="ID2");
```

```
  ID1 x.x x.y
1   1   a   f
2   2   b   g
3   3   c   h
4   4   d   i
5   5   e   j
```

Modifying Datasets: reshaping

- You can use reshape to transform data from wide format to long format and vice versa.
- Suppose you need to reshape a wide dataset into long format and suppose the dataset has variable names t.1 and t.2.
- Use ".1" and ".2" suffixes to make the reshaping much easier.

```
wide <- data.frame(ID=1:5,x=c(10,20,30,40,50),t.1=letters[1:5],
                  t.2=letters[22:26]);
reshape(wide, varying=c("t.1", "t.2"), direction="long");
```

```
    ID  x time t id
1.1  1 10    1 a  1
2.1  2 20    1 b  2
3.1  3 30    1 c  3
4.1  4 40    1 d  4
5.1  5 50    1 e  5
1.2  1 10    2 v  1
2.2  2 20    2 w  2
3.2  3 30    2 x  3
4.2  4 40    2 y  4
5.2  5 50    2 z  5
```

Modifying Datasets: reshaping

- A reshaped dataset contains attributes that make it easy to move between long and wide format, after the first reshape.

```
reshape.long<-reshape(wide, varying=c("t.1", "t.2"), direction="long",
                        timevar="visit", idvar="index", drop="ID");
reshape(reshape.long, direction="wide");
```

```
     x index t.1 t.2
1.1 10     1   a   v
2.1 20     2   b   w
3.1 30     3   c   x
4.1 40     4   d   y
5.1 50     5   e   z
```

Modifying Datasets: reshaping

- Reshape a long dataset into wide format

```
long <- data.frame(ID=rep(1:5, 2), x=rep(c(10,20,30,40,50), 2),
            t=c(letters[1:5], letters[22:26]), time=rep(1:2, each=5));
reshape(long,idvar="ID",v.names="t",timevar="time",direction="wide");

  ID  x t.1 t.2
1  1 10   a   v
2  2 20   b   w
3  3 30   c   x
4  4 40   d   y
5  5 50   e   z

reshape.wide<-reshape(long, idvar="ID", v.names=c("t", "x"),
                      timevar="time", direction="wide");
reshape(reshape.wide, direction="long");

    ID time t  x
1.1  1    1 a 10
2.1  2    1 b 20
3.1  3    1 c 30
4.1  4    1 d 40
5.1  5    1 e 50
1.2  1    2 v 10
2.2  2    2 w 20
3.2  3    2 x 30
4.2  4    2 y 40
```

Modifying Datasets: stacking

- The `stack()` function concatenates multiple vectors from separate columns of a dataframe into a single vector along with a factor indicating where each observation originated; `unstack()` reverses this operation.
- First argument is an object to be stacked or unstacked.
- Second argument is a formula, “values to unstack” $\sim$ “groups to create”

```
data <- data.frame(label=rep(c("A", "B", "C"), 3), value=sample(9));
uns <- unstack(data, value~label);
# Re-stack
stack(uns);

  values ind
1      4   A
2      7   A
3      5   A
4      2   B
5      6   B
6      3   B
7      9   C
8      1   C
9      8   C

# Select which columns to stack
data <- data.frame(A=sample(1:9, 3), B=sample(1:9, 3),
                   C=sample(1:9, 3));
stack(data, select=c("A", "B"));
```

Modifying Datasets: missing obs

- For removing missing data from a data set, use `na.omit()`.
- You can use `na.fail()` to signal an error if a dataset contains `NA`.
- To see which rows have no missing data, use `complete.cases()` which returns a logical vector.

```
data <- data.frame(x=c(1,2,3), y=c(5, NA, 8));
na.omit(data); # Remove all rows with missing data
```

```
  x y
1 1 5
3 3 8
```

```
sum(complete.cases(data)); # Get the number of complete cases
```

```
[1] 2
```

```
sum(!complete.cases(data)); # Get the number of incomplete cases
```

```
[1] 1
```

by()

- A data frame is split by row into data frames subsetted by the values of one or more factors, and function is applied to each subset in turn.

```
tmp <- data.frame(species=c(rep(c(1,2,3), each=5)),
     petal.length=c(rnorm(5,4.5,1),rnorm(5,4.5,1), rnorm(5,5.5,1)),
     petal.width=c(rnorm(5,2.5,1),rnorm(5,2.5,1), rnorm(5,4,1)))

tmp$species <- factor(tmp$species) # make species a factor
tmp
```

```
   species petal.length petal.width
1        1     5.002696   0.3222365
2        1     3.685136   2.3584580
3        1     3.245676   1.3645935
4        1     4.615332   3.1503778
5        1     6.575463   2.8671611
6        2     5.944117   1.2899135
7        2     4.435364   1.8099720
8        2     6.465564   4.1094992
9        2     3.241968   1.7620894
10       2     4.735524   3.4756412
11       3     4.221613   3.7060014
12       3     6.122622   3.4237337
13       3     5.418786   3.3835154
14       3     3.803783   4.7952939
15       3     4.864628   4.8424454
```

by()

```
by(tmp, tmp$species, function(x){
  # caculate the mean petal length for each species
  mean.pl <- mean(x$petal.length)
})

tmp$species: 1
[1] 4.624861
------------------------------------------------
tmp$species: 2
[1] 4.964507
------------------------------------------------
tmp$species: 3
[1] 4.886286
```

ave()

- Splits of a given numeric data are averaged (default), where splits are carried out using grouping variables, typically factors.
- Instead of averaging you can apply your own function FUN.
- Note that the output will have the same length with the given input data.

```
load("wage1000.rda")
names(data)
ave(data$wage, data$female, data$nonwhite)
```

aggregate()

- Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.
- Grouping elements are coerced to factors before use.

```
load("wage1000.rda")
aggregate(data$wage, by=list(data$female, data$nonwhite), FUN=mean)

  Group.1 Group.2         x
1       0       0 15.000501
2       1       0 11.369663
3       0       1 11.521714
4       1       1  9.296579

#try
aggregate(
  data[,c(1,5:7)],by=list(data$female,data$unionmember), function(x){
  tmp <- sum(x, na.rm=T);
  nonwhite_share <- sum(x[data$nonwhite==1], na.rm=T)/tmp;
  white_share <- sum(x[data$nonwhite==0], na.rm=T)/tmp;
  return(c(tmp, nonwhite_share, white_share));
  }
)
```

dplyr

- If you have been a Stata user and now switching to R, this might be the most beneficial package for you.
- It gives you similar functionality to handle/modify data in R.
- It provides simple "verbs", functions that correspond to the most common data manipulation tasks. Here are some examples.

`filter()`	`slice()`
`arrange()`	
`select()`	`rename()`
`dinstinct()`	
`mutate()`	`transmute()`
`summarise()`	
`sample_n()`	`sample_frac()`

dplyr

- `filter()` allows you to select a subset of rows in a data frame.
- `filter()` works similarly to `subset()` except that you can give it any number of filtering conditions, which are joined together with `&` (not `&&` which is easy to do accidentally!).
- `arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them.
- It takes a data frame, and a set of column names (or more complicated expressions) to order by.
- To sort a column in descending order, use `desc()`.
- `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions.

dplyr

```
# Don't forget to install dlplyr first
library(dplyr);
library(nycflights13);
filter(flights, month == 1, day == 1);

# A tibble: 842 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
<int>
1   2013     1     1      517            515         2      830
819
2   2013     1     1      533            529         4      850
830
3   2013     1     1      542            540         2      923
850
4   2013     1     1      544            545        -1     1004
1022
# ... with 838 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour

arrange(flights, desc(arr_delay));
select(flights, year, month, day);
select(flights, year:day);
select(flights,-(year:day));
```

dplyr

- There are a number of helper functions you can use within `select()`, like `starts_with()`, `ends_with()`, `matches()` and `contains()`.
- These let you quickly match larger blocks of variables that meet some criterion.
- `rename()` allows you to rename existing variables.
- `distinct()` allows you to to find unique values in a table.
- `mutate()` allows you to generate new variables.
- `summarise()` is useful for generating summary statistics.
- Note that `summarize()` does not exist and would give an error message.

dplyr

```
rename(flights, tail_num = tailnum);
distinct(flights, tailnum);
mutate(flights,
       gain = arr_delay - dep_delay,
       gain_per_hour = gain / (air_time / 60));
summarise(flights,
          delay = mean(dep_delay, na.rm = TRUE));
```

dplyr

- One of the most useful attributes of `dplyr` is its grouped operators.
- For example, `group_by()` splits a dataset into specified groups of rows.
- Then you can apply the verbs above on the resulting object and they will be automatically applied "by group."
- When you group by multiple variables, each summary peels off one level of the grouping.

`select()`	is the same as ungrouped `select()`
`arrange()`	orders first by the grouping variables
`mutate() filter()`	are most useful in conjunction with window functions. `functions")`.
`slice()`	extracts rows within each group
`summarise()`	is powerful and easy to understand, as described in more detai

dplyr

- The following example splits the complete dataset into individual planes and then summarises each plane by counting the number of flights `(count = n())` and computing the average distance `(dist = mean(Distance, na.rm = TRUE))` and arrival delay (`delay = mean(ArrDelay, na.rm = TRUE)`).

```
by_tailnum <- group_by(flights, tailnum);
delay <- summarise(by_tailnum,
                   count = n(),
                   dist = mean(distance, na.rm = TRUE),
                   delay = mean(arr_delay, na.rm = TRUE));
delay <- filter(delay, count > 20, dist < 2000);
```

dplyr

- A useful functionality of dplyr is the fact that it allows for chaining. To understand this take a look at below chunk.

```
a1 <- group_by(flights, year, month, day);
a2 <- select(a1, arr_delay, dep_delay);
a3 <- summarise(a2,
                arr = mean(arr_delay, na.rm = TRUE),
                dep = mean(dep_delay, na.rm = TRUE));
a4 <- filter(a3, arr > 30 | dep > 30);
```

- You had to save all intermediate output. If you don't want to save the intermediate results, you need to wrap the function calls inside each other.
- This can generate difficult to read codes, you can lose the track of parantheses.

```
filter(
  summarise(
    select(
      group_by(flights, year, month, day),
      arr_delay, dep_delay
    ),
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ),
  arr > 30 | dep > 30
)
```

dplyr

- To get around this problem, dplyr is compatible with the pipe, %>%, from the magrittr package.
- `x %>% f(y)` turns into `f(x, y)` so you can use it to rewrite multiple operations that you can read left-to-right, top-to-bottom.

```
flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(arr > 30 | dep > 30)
```

data.table

- For a quick introduction to data.table package, see https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html

Plotting

- R allows you to create high quality graphics.
- You can use base functions or a more sophisticated approach using `ggplot2`.
- The base graphing functions consists of two groups: high-level and low-level functions.
- High-level functions create new plots and low-level functions add information to an existing plot.

Plotting

High-level plot functions

`plot()`	Scatterplot
`hist()`	Histogram
`boxplot()`	Boxplot
`qqplot(), qqnorm(), qqline()`	Qunatile plots
`sunflowerplot()`	Sunfower scatterplot
`pairs()`	Scatter plot matrix
`symbols()`	Draw symbols on a plot
`dotchart(), barplot(), pie()`	Dot chart, bar chart, pie chart
`curve()`	Draw a curve from a given function
`image()`	Create a grid of colored rectangles
`contour(), filled.contour()`	Contour plot
`persp()`	Plot 3-D surface

Plotting

Low-level plot functions	
`points()`	Add points to a figure
`lines()`	Add lines to a figure
`text()`	Insert text in the plot region
`mtext()`	Insert text in the figure and outer margins
`title()`	Add figure title or outer title
`legend()`	Insert legend
`axis()`, `axis.Date()`	Customize axes
`abline()`	Add horizontal and vertical lines or a single line
`box()`	Draw a box around the current plot
`rug()`	Add a 1-D plot of the data to the figure
`polygon()`	Draw a polygon
`rect()`	Draw a rectangle
`arrows()`	Draw arrows
`segments()`	Draw line segments
`trans3d()`	Add 2-D components to a 3-D plot

Plotting

`plot()` function is used for producing scatterplots and line graphs. Common arguments for plot() are

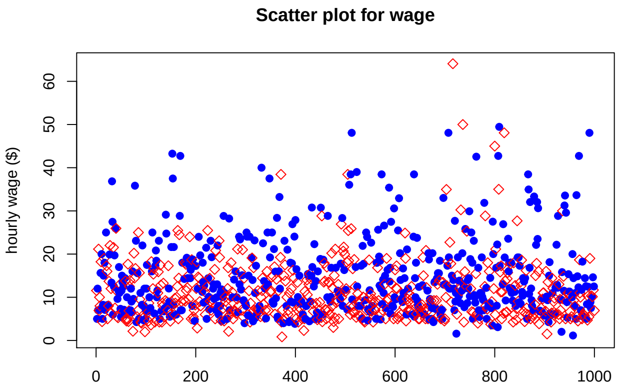
`type`	1-character string denoting the plot type
`xlim`	x limits, c(x1, x2)
`ylim`	y limits, c(y1, y2)
`log`	Character string that contains "x" if x-axis is log-scale, "y" if y-axis is log scale, and "xy" if both axes are log scale
`main`	Main title for the plot
`sub`	Sub title for the plot
`xlab`	x-axis label
`ylab`	y-axis label
`ann`	Logical, should default annotation appear on plot
`axes`	Logical, should both axes be drawn
`col`	Color for lines and points, either a character string or a number that indexes the `palette()`
`pch`	Number referencing a plotting symbol or a character string
`cex`	A number giving the character expansion of the plot symbols
`lty`	Number referencing a line type
`lwd`	Line width

Plotting: Scatter plots

```
#Graphics
x <- rnorm(50)
y <- rnorm(50)
group <- rbinom(50, size=1, prob=.5)
# Basic Scatterplot
plot(x, y)
plot(x, y, xlab="X", ylab="Y", main="Y vs X", pch=1, col="red")
# Distinguish between two separate groups
plot(x, y, xlab="X", ylab="Y", main="Y vs X",
     pch=ifelse(group==1, 1, 19),
     col=ifelse(group==1, "red", "blue"))
# Type colors() to see the list of colors
# The points argument can be, (1) two separate vectors: one
# is the x-coordinates and the other is the y-coordinates
plot(x, y, xlab="X", ylab="Y", main="Y vs X", type="n")
points(x[group==1], y[group==1], pch=1, col="red")
points(x[group==0], y[group==0], pch=19, col="blue")
#Alternatively
plot(x, y, xlab="X", ylab="Y", main="Y vs X", type="n")
points(cbind(x,y)[group==1,], pch=1, col="red")
points(cbind(x,y)[group==0,], pch=19, col="blue")
```

Plotting

```
load('wage1000.rda')
head(data)
wage=data$wage
plot(wage,ylab="hourly wage ($)",  main="Scatter plot for wage",
     ylim=c(min(wage),max(data$wage)),
     pch=ifelse(female==1,5,19),
     col=ifelse(female==1,"red","blue"));
```



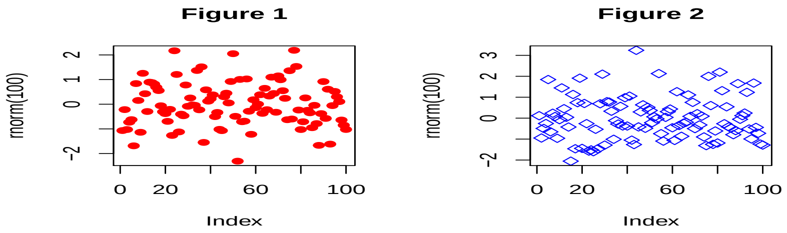
Plotting: Line and histogram plots

```
# Basic Line Graphs
plot(sort(x), sort(y), type="l", lty=2, lwd=2, col="blue")
# Alternatively
plot(x, y, type="n")
lines(sort(x), sort(y), type="b")
lines(cbind(sort(x),sort(y)), type="l", lty=1, col="blue")
#if there is only one component, the argument is plotted against
#its index (same with plot and points)
plot(sort(x), type="n")
lines(sort(x), type="b", pch=8, col="red")
lines(sort(y), type="l", lty=6, col="blue")

# Basic Histogram
hist(x, main="Histogram of X", col="deeppink4")
# Plot histogram along with a normal density
# Set freq=FALSE, so that the density histogram is plotted
hist(x,freq=FALSE,col="red",main="Histogram with Normal Curve")
# Uses the observed mean and standard deviation for the curve
xpts <- seq(min(x), max(x), length=50)
ypts <- dnorm(xpts, mean=mean(x), sd=sd(x))
lines(xpts, ypts, lwd=2,col="blue")
```

Plotting: Multiple graphs

- To create a $n \times m$ grid of figures use `par()` with either the `mfcol` or `mfrow` settings.
  - `mfcol=c(nr, nc)` adds figures by column
  - `mfrow=c(nr, nc)` adds figures by row
- To create a more complex arrangement of multiple plots, check `?layout()`.

```
# Figure with two plots side by side
par(mfrow=c(1,2))
plot(rnorm(100), main="Figure 1", pch=19, col="red")
plot(rnorm(100), main="Figure 2", pch=5, col="blue")
par(mfrow=c(1,1))
```



Plotting: curve()

- The function curve() draws a curve corresponding to a given function.

```
# Curve function
# Plot a 5th order polynomial
curve(3*x^5-5*x^3+2*x,from=-1.25,to=1.25,lwd=2,col="blue")
# Plot the gamma density
curve(dgamma(x,shape=2,scale=1),from=0, to=7,lwd=2,col="red")
# Plot multiple curves: the first curve determines the x-axis
curve(dnorm, from=-3, to=5, lwd=2, col="red")
curve(dnorm(x, mean=2), lwd=2, col="blue", add=TRUE)
# Add vertical lines at the means
lines(c(0, 0), c(0, dnorm(0)), lty=2, col="red")
lines(c(2, 2), c(0, dnorm(2, mean=2)), lty=2, col="blue")
```

Plotting: `legend()`

- Legends are added to a figure using `legend()`, legends can be added to the plot region, figure margin, or the outer margin.

```
# Plot 1
set.seed(789)
x1 <- rnorm(50)
x2 <- rnorm(50, mean=2)
y1 <- rnorm(50)
y2 <- rnorm(50, mean=2)
# Use range to determine a plot region that is large enough
plot(range(x1,x2),range(y1,y2),main="Figure 1",type="n",
        xlab="X",ylab="Y")
points(x1, y1, col="red", pch=19) # Group 1
points(x2, y2, col="blue", pch=1) # Group 2
legend("topleft", c("Group 1","Group 2"), pch=c(19,1),
       col=c("red", "blue"), horiz=TRUE, bty="n")
# PLOT 2
plot(range(x1,x2),range(y1,y2),main="Figure 2",type="n",
        xlab="X",ylab="Y")
lines(sort(x1), sort(y1), col="red", type="o", pch=19) # Group 1
lines(sort(x2), sort(y2), col="blue", type="o", pch=1) # Group 2
legend(-3, 3, c("Group 1", "Group 2"), pch=c(19,1),
        col=c("red", "blue"),horiz=TRUE, bty="o", lty=1)
```

Plotting: axis()

- The functions axis() and axis.Date() are used create custom axes.

```
# Plot with no axes
par(mar=c(5,5,5,5))
plot(1:10, axes=FALSE, ann=FALSE)
# Add an axis on side 2 (left)
axis(2)
# Add an axis on side 3 (top), specify tick mark location,
# and add labels
axis(3, at=seq(1,10,by=.5),
         labels=format(seq(1,10,by=.5), nsmall=3))
# Add an axis on side 4 (right), specify tick mark location
# and rotate labels
axis(4, at=1:10, las=2) # las: style of axis label
# Add axis on side 1 (bottom), with labels rotated 45 degrees
tck <- axis(1, labels=FALSE)
text(tck, par("usr")[3]-1, labels=paste("Label", tck),
         srt=45, adj=1,xpd=TRUE)
box() # Add box aroung plot region
# Add axis labels
mtext(paste("Side", 1:4), side=1:4, line=3.5, font=2)
```

Plotting: `axis.Date()`

- The functions axis() and axis.Date() are used create custom axes.

```
#axis.Date
dates <- seq.Date(Sys.Date(),by="3 day", length.out=25)
y <- sort(rexp(length(dates)))
plot(dates, y, xlab="Date", ylab="Y", main="Y vs Date",
     axes=FALSE, type="o", pch=18, col="darkorange4", cex=1.5)
# Y-axis
axis(2, at=seq(0, max(y), by=.5))
# X-axis
axis.Date(1, at=seq.Date(min(dates), max(dates), "week"),
          format="%b %d\n%Y", padj=.5)
axis.Date(1, at=seq.Date(min(dates), max(dates), "day"),
          labels=FALSE, tcl=-.25)
box()
```

ggplot2

- ggplot2 is an R package for producing statistical, or data, graphics, but it is unlike most other graphics packages because it has a deep underlying grammar.
- This makes ggplot2 very powerful, because you are not limited to a set of pre-specified graphics, but you can create new graphics that are precisely tailored for your problem.
- The grammar tells us that a statistical graphic is a mapping from data to aesthetic attributes (colour, shape, size) of geometric objects (points, lines, bars).

ggplot2

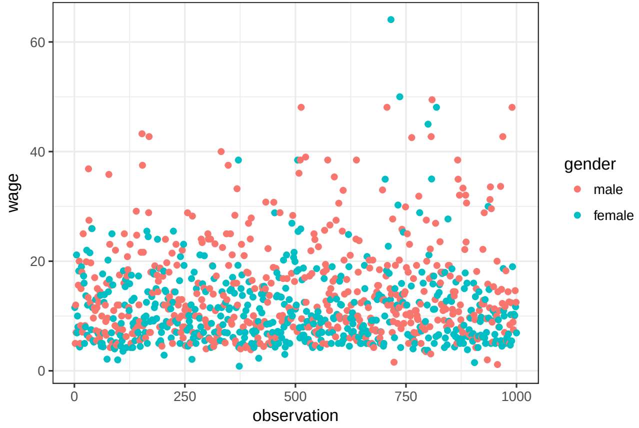
- The data that you want to visualise and a set of aesthetic mappings describing how variables in the data are mapped to aesthetic attributes that you can perceive.
- Geometric objects, geoms for short, represent what you actually see on the plot: points, lines, polygons, etc.
- Statistical transformations, stats for short, summarise data in many useful ways.
- The scales map values in the data space to values in an aesthetic space, whether it be colour, or size, or shape. Scales draw a legend or axes, which provide an inverse mapping to make it possible to read the original data values from the graph.
- A coordinate system, coord for short, describes how data coordinates are mapped to the plane of the graphic. It also provides axes and gridlines to make it possible to read the graph.
- A faceting specification describes how to break up the data into subsets and how to display those subsets as small multiples.

ggplot2

```
ggplot(data = <default data set>,
     aes(x = <default x axis variable>,
         y = <default y axis variable>,
         ... <other default aesthetic mappings>),
     ... <other plot defaults>) +

     geom_<geom type>(aes(size = <size variable for this geom>,
                          ... <other aesthetic mappings>),
                data = <data for this point geom>,
                stat = <statistic string or function>,
                position = <position string or function>,
                color = <"fixed color specification">,
                <other arguments, possibly passed to the _stat_ function) +

scale_<aesthetic>_<type>(name = <"scale label">,
                   breaks = <where to put tick marks>,
                   labels = <labels for tick marks>,
                   ... <other options for the scale>) +

theme(plot.background = element_rect(fill = "gray"),
      ... <other theme elements>)
```

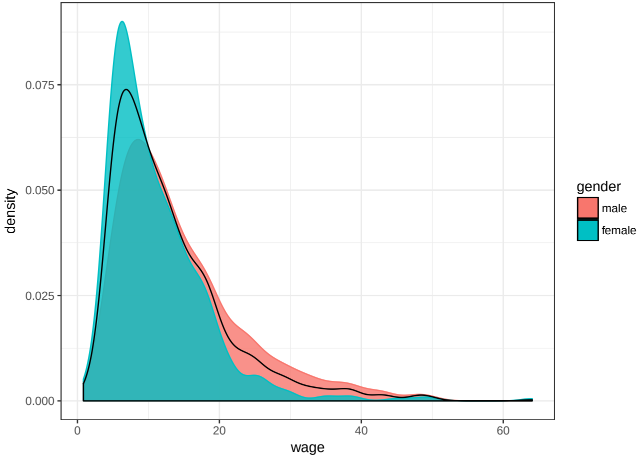
Plotting

```
data$gender <- factor(data$female, levels=c(0,1),
                      labels=c("male","female"));
data$obs <- 1:nrow(data)
ggplot(data, aes(x=obs, y=wage, color=gender))+
   geom_point() + theme_bw() + labs(x="observation")
```



Plotting

```
ggplot() +
 geom_density(aes(wage, color=gender, fill=gender), data, alpha=0.8) +
 geom_density(aes(wage), data) + theme_bw();
```



R-bloggers

- Here are some examples from R-bloggers:
  - Pop singers' vocal frequency: http://giorasimchoni.com/2017/12/10/2017-12-10-ave-mariah/
  - Crime in downtown Houston: https://github.com/tidyverse/ggplot2/wiki/Crime-in-Downtown-Houston,-Texas-:-Combining-ggplot2-and-Google-Maps
  - Hurricane exposure package: https://cran.rstudio.com/web/packages/hurricaneexposure/vignettes/hurricaneexposure.html
  - Mapping election results: https://www.r-bloggers.com/mapping-election-results-with-r-and-choroplethr/
  - Mapping census data: https://arilamstein.com/blog/2015/06/25/learn-to-map-census-data-in-r/

Regression analysis

- Regression analysis is mainly carried out using `lm()` or `glm()`.
- Both use a convenient formula syntax to specify the form of the statistical model to be fit.
- A formula takes the following form: `response var` $\sim$ `explanatory var`

```
fit <- lm(Y ~ X)
```

- Additional explanatory variables can be included using the "+" symbol.

```
fit <- lm(Y ~ X + Z)
```

Regression analysis

- The below table lists use of common symbols in an R modeling formula.

Symbol	Example	
+	$+X$	include X
-	$-X$	delete X
:	$X : Z$	include the interaction between X and Z
*	$X * Z$	include X, Z and their interaction
|	$X\|Z$	include X given Z
^	$(X + Z + W)\hat{}3$	include X, Z, W and all interactions up to three way
I	$I(X * Z)$	as is: include multiplication of X and Z.
1	$-1 + X$	do not include an intercept

Wage Regression

- The data set is wage1000.rda without headers.
- It is wage data with 1000 observations from the US Bureau of Census Current Population survey, March 1995.
- The underlying population is the employed labor force, age 18-65. The variables are as follows:

  1. hourly wage
  2. female (1= worker = female)
  3. non-white (1= worker = non-white)
  4. union (1 = worker = unionized)
  5. education (years of education)
  6. experience (years of work experience)
  7. age

Wage Regression

- attach the data frame to the search path so we can work with each variable individually.
- Create a table of decriptives.

```
load('wage1000.rda') # load the data
attach(data)
stargazer(data, title="Descriptive Statistics", type="text")
```

```
Descriptive Statistics
=============================================
Statistic      N     Mean   St. Dev.  Min    Max
---------------------------------------------
wage        1,000 12.817  8.244    0.840 64.080
female      1,000 0.491   0.500      0      1
nonwhite    1,000 0.146   0.353      0      1
unionmember 1,000 0.164   0.370      0      1
education   1,000 13.183  2.865      0     20
experience  1,000 19.235  11.829     0     56
age         1,000 38.418  11.698    18     65
---------------------------------------------
```

Wage Regression: lm()

- Use lm() function to run regressions.

```
# Use lm function for regression
ols1 <- lm(wage ~ female+nonwhite+unionmember
                +education+experience)
names(ols1)
ols1$coefficients
coefficients(ols1)
ols1$coefficients["female"]
ols1$residuals
residuals(ols1)
ols1$fitted.values #predicted values
fitted(ols1)
# Print results by summary function
summary(ols1)
names(summary(ols1))
summary(ols1)$adj.r.squared
stargazer(ols1, title="OLS 1", type="text")
```

Wage Regression: lm()

- Use lm() function to run regressions.

```
# Run another model by adding square of experience
ols2 <- lm(wage ~ female+nonwhite+unionmember+education+experience
           +I(experience^2))
names(ols2)
ols2$coefficients
coefficients(ols2)
ols2$coefficients["female"]
ols2$residuals
residuals(ols2)
ols2$fitted.values #predicted values
fitted(ols2)
# Print results by summary function
summary(ols2)
names(summary(ols2))
summary(ols2)$adj.r.squared
stargazer(ols2, title="OLS 2", type="text")
```

Wage Regression: `lm()`

- Print results in table by using `stargazer()`:

```r
stargazer(ols1,ols2,title="Results",type="text")
```

```
====================================================================
                           Dependent variable: wage
                        (1)                       (2)
--------------------------------------------------------------------
female               -3.099***                 -3.026***
                      (0.424)                   (0.419)
nonwhite             -1.607***                 -1.553***
                      (0.603)                   (0.597)
unionmember            0.821                     0.741
                      (0.583)                   (0.577)
education             1.498***                  1.446***
                      (0.075)                   (0.075)
experience            0.170***                  0.452***
                      (0.018)                   (0.062)
I(experience2)                                 -0.007***
                                                (0.001)
Constant             -8.579***                 -9.960***
                      (1.161)                   (1.184)
--------------------------------------------------------------------
Observations           1,000                     1,000
R2                     0.348                     0.363
Adjusted R2            0.345                     0.359
====================================================================
Note:                                   *p<0.1; **p<0.05; ***p<0.01
```

Example via Simulated Data

- Assume that hourly wages in your general population of interest follow different distributions for female and male workers.
- Assume female wages follow a normal distribution with mean $\$20$ and standard deviation $\$3$,
- Assume male wages follow a normal distribution with mean $\$30$ and standard deviation $\$5$.
- Generate $1e3$ draws each from these distributions.

Example via Simulated Data

```
N <- 1000;
fmean <- 20;
fstd <- 3;
fy <- matrix(rnorm(N, fmean, fstd), N);
mmean <- 30;
mstd <- 5;
my <- matrix(rnorm(N, mmean, mstd),N);
male <- rbind(matrix(0, N), matrix(1,N));
gender <- factor(male, levels=c(0,1), labels=c("female", "male"));
data <- data.frame(wage=rbind(fy,my), gender=gender);
```

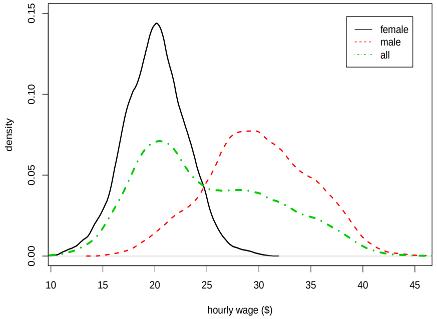
Example via Simulated Data

- Plot the two samples using kernel densities (think of it a s a smooth histogram).
- First, we need to generate density estimates for each data point. Obtain kernel density estimates, using Epanechnikov method.

```
ally <- rbind(fy,my);
fdens <- density(fy, kernel="epanechnikov", n=1000);
mdens <- density(my, kernel="epanechnikov", n=1000);
alldens <- density(ally, kernel="epanechnikov", n=2000);
```

Example via Simulated Data

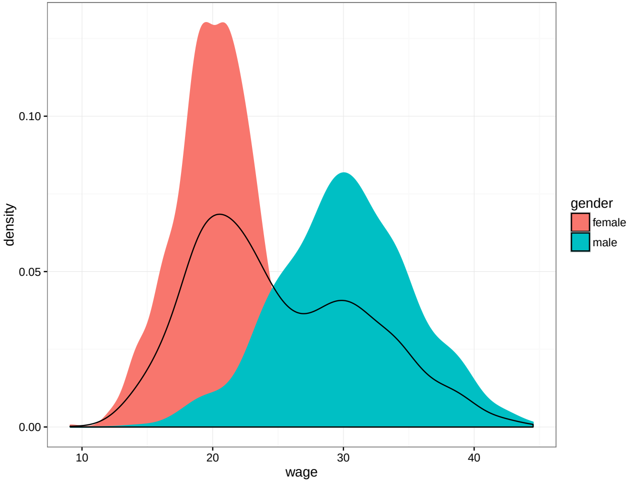
```
plot(fdens, type="l", main="", xlab="hourly wage", ylab = "density",
     xlim=c(min(ally), max(ally)), ylim=c(0,0.15), lwd=2);
lines(mdens, col=2, lty=2, lwd=2);
lines(alldens, col=3, lty=4, lwd=3);
labels<-c("female", "male", "all");
legend("topright", inset=.05, labels,
       lwd=1, lty=c(1,2,4), col=c(1,2,3));
```



Example via Simulated Data

- Plot empirical densities of the two samples along with the entire sample.

```
ggplot() + geom_density(aes(wage,color=gender,fill=gender),data) +
  geom_density(aes(wage),data) + theme_bw();
```



Example via Simulated Data

- generate a table with basic descriptive statistics:

```
attach(data);
stargazer(data,title="Descriptive Statistics",type="text");
```

```
Descriptive Statistics
============================================
Statistic   N    Mean  St. Dev.  Min    Max
--------------------------------------------
wage      2,000 25.138  6.540   9.105 44.508
--------------------------------------------
```

- Suppose you just want to estimate the mean, so the only explanatory variable is a vector of ones.

```
ols <- lm(wage ~ 1);
stargazer(ols, title="Results", type="text");
```

```
Results
===============================================
                        Dependent variable:
                    ---------------------------
                               wage
-----------------------------------------------
Constant                     25.138***
                              (0.146)


-----------------------------------------------
Observations                   2,000
R2                             0.000
Adjusted R2                    0.000
Residual Std. Error      6.540 (df = 1999)
===============================================
Note:                    *p<0.1; **p<0.05; ***p<0.01
```