Introduction to R: **Data Management**

Day 2, Part A

In this lecture

1. Loading data
2. Renaming variables
3. Sorting
4. Adding additional rows
5. Adding additional columns
6. Saving data
7. Managing the work space

Loading data

Data can come in many file formats, but the ones you are likely to encounter most often at IHME are:

- **.rdata** or **.rds** - R's file format
- **.csv** - Comma delimited files
- **.dta** - Stata files

Loading data

.rdata files may contain any type of R object and may have more than one object.

The data in these files are loaded using the `load()` function, which will put all of the stored objects directly into the work space with names already assigned:

```
> load("C:/Users/ngraetz/Documents/repos/r_training_penn/data/us_state_cigarett
+     verbose = T)
Loading objects:
  cig_tax
  cig_csmp
  pop
  locs
> class(cig_tax)
[1] "data.frame"
> class(cig_csmp)
[1] "data.frame"
> class(pop)
[1] "data.frame"
> class(locs)
[1] "data.frame"
```

Loading data

.rds files may contain any type of R object but can only have a single object.

The data in a .rds file are loaded using the `readRDS()` function. The data loaded using this function must be assigned to an object:

```
> zmb <- readRDS("C:/Users/ngraetz/Documents/repos/r_training_penn/data/zmb_mcp
> str(zmb)
'data.frame':   1512 obs. of  14 variables:
 $ province   : chr  "central" "central" "central" "central" ...
 $ district   : chr  "chibombo" "chibombo" "chibombo" "chibombo" ...
 $ year       : int  1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 ...
 $ q5         : num  149 148 147 145 144 ...
 $ anc1       : num  0.988 0.989 0.989 0.989 0.989 ...
 $ sba        : num  0.404 0.374 0.345 0.316 0.291 ...
 $ polio      : num  0.951 0.931 0.901 0.858 0.804 ...
 $ measles    : num  0.976 0.978 0.978 0.974 0.966 ...
 $ dpt3       : num  0.974 0.967 0.956 0.94 0.915 ...
 $ ebf        : num  0.0136 0.0182 0.0257 0.0381 0.0581 ...
 $ itn        : num  0.011 0.0105 0.0105 0.0108 0.0114 ...
 $ irs        : num  0.0253 0.0186 0.0146 0.0119 0.0101 ...
 $ electricity: num  0.0579 0.0584 0.0578 0.0561 0.0535 ...
 $ female_edu : num  4.08 4.17 4.26 4.35 4.43 ...
```

Loading data

.csv files contain tabular data that are usually loaded as a data frame with the `read.csv()` function:

```
> mmr <- read.csv("C:/Users/ngraetz/Documents/repos/r_training_penn/data/mmr_da
> str(mmr)
'data.frame':   24 obs. of  7 variables:
 $ year_id           : int  2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 .
 $ mmr               : num  52.6 50.4 25.6 61.3 25.2 ...
 $ maternal_education: num  9.9 5.94 11.54 14.7 14.64 ...
 $ ldi               : num  10594 2774 20783 33327 40454 ...
 $ location_name     : Factor w/ 24 levels "Afghanistan",..: 9 6 15 13 2 23 8 2
 $ super_region_name : Factor w/ 6 levels "High-income",..: 5 5 5 1 1 1 1 1 2 2
 $ region_name       : Factor w/ 15 levels "Australasia",..: 4 10 10 6 1 14 11
```

Loading data

By default, `read.csv()` converts anything that's not obviously a numeric or logical to a factor. Often, we want these variables to be characters instead, so to change this behavior, set the `stringsAsFactors` option to `FALSE`:

```
> mmr <- read.csv("C:/Users/ngraetz/Documents/repos/r_training_penn/data/mmr_da
+     stringsAsFactors = F)
> str(mmr)
'data.frame':   24 obs. of  7 variables:
 $ year_id           : int  2015 2015 2015 2015 2015 2015 2015 2015 2015 2015 .
 $ mmr               : num  52.6 50.4 25.6 61.3 25.2 ...
 $ maternal_education: num  9.9 5.94 11.54 14.7 14.64 ...
 $ ldi               : num  10594 2774 20783 33327 40454 ...
 $ location_name     : chr  "China" "Cambodia" "Malaysia" "Japan" ...
 $ super_region_name : chr  "Southeast Asia, East Asia, and Oceania" "Southeast
 $ region_name       : chr  "East Asia" "Southeast Asia" "Southeast Asia" "High
```

Loading data

.dta files also contain tabular data that are usually loaded as a data frame.

- Older .dta files (Stata versions 5-12) can be loaded using `read.dta()` in the `foreign` library.
- Newer .dta files can be loaded using `readstata13()` in the `readstata13` library or `read_dta()` in the `haven` library.
- All of these functions operate similarly to `read.csv()`.

Loading data

.dta files also contain tabular data that are usually loaded as a data frame.

- Older .dta files (Stata versions 5-12) can be loaded using `read.dta()` in the `foreign` library.
- Newer .dta files can be loaded using `readstata13()` in the `readstata13` library or `read_dta()` in the `haven` library.
- All of these functions operate similarly to `read.csv()`.

Note that the `foreign` and `haven` libraries also include functions for reading data files from other statistical software (e.g., SPSS, SAS).

File paths

File paths must have only forward slashes:

- RIGHT: `"C:/path/to/your/directory/"`
- WRONG: `"C:\path\to\your\directory\"`

File paths

File paths must have only forward slashes:

- RIGHT: `"C:/path/to/your/directory/"`
- WRONG: `"C:\path\to\your\directory\"`

And they can be either absolute:

```
> data <- read.csv("C:/Users/ngraetz/Documents/repos/r_training_penn/data/educa
```

or relative to the current directory:

```
> data <- read.csv("data/education_2015.csv")
```

File paths

Absolute file paths are recommended for loading and saving data! They will always work, whereas relative paths will only work if you are in the correct directory.

File paths

Absolute file paths are recommended for loading and saving data! They will always work, whereas relative paths will only work if you are in the correct directory.

It's possible to manually set a working directory, which guarantees that you are in the correct place (at least temporarily):

```
> setwd("C:/Users/ngraetz/Documents/repos/r_training_penn/")
> data <- read.csv("data/education_2015.csv")
```

BUT `setwd()` interacts badly with certain advanced R features, so absolute file paths for data are still recommended.

File paths

If you're loading multiple files from the same directory, it can be useful to store that directory as its own object and then use the `paste0()` command to create full file paths:

```
> main_dir <- "C:/Users/ngraetz/Documents/repos/r_training_penn/"
> data <- read.csv(paste0(main_dir, "data/education_2015.csv"))
```

Renaming variables in a data frame

We can use the `names()` function to view the column names (variables) in a data frame:

```
> data <- read.csv(paste0(main_dir, "data/CPI_2015_EU.csv"),
+                  stringsAsFactors=F)
> names(data)
 [1] "Rank"                                  "CPI2015"
 [3] "Country"                               "Region"
 [5] "wbcode"                                "World.Bank.CPIA"
 [7] "World.Economic.Forum.EOS"              "Bertelsmann.Foundation.TI"
 [9] "African.Dev.Bank"                      "IMD.World.Competitiveness.Yearbook
[11] "Bertelsmann.Foundation.SGI"            "World.Justice.Project.ROL"
[13] "PRS.International.Country.Risk.Guide" "Economist.Intelligence.Unit"
[15] "IHS.Global.Insight"                    "PERC.Asia.Risk.Guide"
[17] "Freedom.House.NIT"                     "CPI2015.2."
[19] "Rank2"                                 "Number.of.Sources"
```

Renaming variables in a data frame

We can also use `names()` to change the column names in a data frame:

```
> names(data) <- c("Rank", "CPI2015", "Country", "Region", "wbcode",
+     "CPIA", "EOS", "TI", "ADB", "IMD", "SGI", "ROL", "PRS", "EIU",
+     "IHS", "PERC", "NIT", "CPI_2", "Rank_2", "N")
> names(data)
 [1] "Rank"    "CPI2015" "Country" "Region"  "wbcode"
 [6] "CPIA"    "EOS"     "TI"      "ADB"     "IMD"
[11] "SGI"     "ROL"     "PRS"     "EIU"     "IHS"
[16] "PERC"    "NIT"     "CPI_2"   "Rank_2"  "N"
```

Renaming variables in a data frame

The output of `names()` is a vector, so you can subset it in the same way as a vector to rename only select columns:

```
> names(data)[5] <- "WB_code"
> names(data)
 [1] "Rank"    "CPI2015" "Country" "Region"  "WB_code"
 [6] "CPIA"    "EOS"     "TI"      "ADB"     "IMD"
[11] "SGI"     "ROL"     "PRS"     "EIU"     "IHS"
[16] "PERC"    "NIT"     "CPI_2"   "Rank_2"  "N"
```

```
> names(data)[names(data) == "CPI_2"] <- "CPI2015_2"
> names(data)
 [1] "Rank"      "CPI2015"   "Country"   "Region"
 [5] "WB_code"   "CPIA"      "EOS"       "TI"
 [9] "ADB"       "IMD"       "SGI"       "ROL"
[13] "PRS"       "EIU"       "IHS"       "PERC"
[17] "NIT"       "CPI2015_2" "Rank_2"    "N"
```

Renaming variables in a data frame

Several packages provide other functions for renaming variables in a data frame that may be more convenient and intuitive:

```
> library(plyr)
> data <- rename(data, c(Country = "country", Region = "region"))
> names(data)
 [1] "Rank"       "CPI2015"    "country"    "region"
 [5] "WB_code"    "CPIA"       "EOS"        "TI"
 [9] "ADB"        "IMD"        "SGI"        "ROL"
[13] "PRS"        "EIU"        "IHS"        "PERC"
[17] "NIT"        "CPI2015_2"  "Rank_2"     "N"
```

Renaming variables in a data frame

Several packages provide other functions for renaming variables in a data frame that may be more convenient and intuitive:

```
> library(dplyr)
> data <- rename(data, rank1 = Rank, rank2 = Rank_2)
> names(data)
 [1] "rank1"      "CPI2015"    "country"    "region"
 [5] "WB_code"    "CPIA"       "EOS"        "TI"
 [9] "ADB"        "IMD"        "SGI"        "ROL"
[13] "PRS"        "EIU"        "IHS"        "PERC"
[17] "NIT"        "CPI2015_2"  "rank2"      "N"
```

Quick note about masking. . .

Different functions in different packages may have the same name. In these cases, the order in which you load the packages matters: whatever is loaded last will 'mask' whatever was loaded earlier.

```
> library(dplyr)

Attaching package: 'dplyr'
The following object is masked from 'package:gridExtra':

    combine
The following object is masked from 'package:ggplot2':

    vars
The following objects are masked from 'package:plyr':

    arrange, count, desc, failwith, id, mutate,
    rename, summarise, summarize
The following objects are masked from 'package:data.table':

    between, first, last
The following objects are masked from 'package:stats':

    filter, lag
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

Quick note about masking. . .

Alternatively, you can specify the package via `package::function()` to be explicit about which function you want to use:

```
> plyr::rename(data, c(Country = "country", Region = "region"))
> dplyr::rename(data, rank1 = Rank, rank2 = Rank_2)
```

This syntax can also be used to access functions without loading the entire library (i.e., the code above would work even if you had not already run `library(plyr)` and `library(dplyr)`).

Sorting data frames

The `order()` function gives you the indices of a vector according to the rank order of the values of that vector:

```
> data$country
 [1] "Denmark"        "Finland"        "Sweden"
 [4] "Netherlands"    "Norway"         "Switzerland"
 [7] "Germany"        "Luxembourg"     "United Kingdom"
[10] "Iceland"        "Belgium"        "Austria"
[13] "Ireland"        "Estonia"        "France"
[16] "Portugal"       "Poland"         "Cyprus"
[19] "Lithuania"      "Slovenia"       "Spain"
[22] "Czech Republic" "Malta"          "Latvia"
[25] "Croatia"        "Hungary"        "Slovakia"
[28] "Greece"         "Romania"        "Italy"
[31] "Bulgaria"
> order(data$country)
 [1] 12 11 31 25 18 22  1 14  2 15  7 28 26 10 13 30 24 19
[19]  8 23  4  5 17 16 29 27 20 21  3  6  9
```

Sorting data frames

This can then be used to reorder the rows of a data frame by index:

```
> data <- data[order(data$country), ]
> head(data)
   rank1 CPI2015         country region WB_code CPIA EOS TI
12    16      76         Austria  WE/EU      AUT   NA  77 NA
11    15      77         Belgium  WE/EU      BEL   NA  79 NA
31    69      41        Bulgaria  WE/EU      BGR   NA  38 53
25    50      51         Croatia  WE/EU      HRV   NA  46 62
18    32      61          Cyprus  WE/EU      CYP   NA  55 NA
22    37      56 Czech Republic  WE/EU      CZE   NA  53 66
   ADB IMD SGI ROL PRS EIU IHS PERC NIT CPI2015_2 rank2 N
12  NA  70  81  81  79  71  73   NA  NA         76    16 7
11  NA  77  81  76  79  71  73   NA  NA         77    15 7
31  NA  32  49  32  41  38  42   NA  47         41    69 9
25  NA  41  57  50  50  54  52   NA  50         51    50 9
18  NA  NA  49  NA  69  71  63   NA  NA         61    32 5
22  NA  44  57  59  50  54  63   NA  55         56    37 9
```

Sorting data frames

order() can also be used to reorder on several variables in sequence:

```
> data <- data[order(data$region, data$CPI2015), ]
> head(data)
   rank1 CPI2015  country region WB_code CPIA EOS TI ADB
31    69      41 Bulgaria  WE/EU     BGR   NA  38 53  NA
30    61      44    Italy  WE/EU     ITA   NA  47 NA  NA
28    58      46   Greece  WE/EU     GRC   NA  45 NA  NA
29    58      46  Romania  WE/EU     ROM   NA  36 62  NA
25    50      51  Croatia  WE/EU     HRV   NA  46 62  NA
26    50      51  Hungary  WE/EU     HUN   NA  47 53  NA
   IMD SGI ROL PRS EIU IHS PERC NIT CPI2015_2 rank2 N
31  32  49  32  41  38  42   NA  47        41    69 9
30  35  49  54  41  38  42   NA  NA        44    61 7
28  43  57  50  41  21  63   NA  NA        46    58 7
29  37  57  45  41  38  42   NA  52        46    58 9
25  41  57  50  50  54  52   NA  50        51    50 9
26  34  41  44  50  71  63   NA  52        51    50 9
```

Adding additional rows

Two data frames with the same columns (variables) can be combined using `rbind()`. This is useful for combining data sets that are formatted similarly but with different contents (e.g., where data are stored in separate files by country, or gender)

```
> cpi_ame <- read.csv(paste0(main_dir, "data/CPI_2015_AME.csv"))
> nrow(cpi_ame)
[1] 26
> cpi_eu <- read.csv(paste0(main_dir, "data/CPI_2015_EU.csv"))
> nrow(cpi_eu)
[1] 31
>
> all <- rbind(cpi_ame, cpi_eu)
> nrow(all)
[1] 57
```

Adding additional rows

`rbind()` requires that all columns in both data frames be named the same:

```
> names(cpi_eu)[1] <- "rank"  # rename for demonstration purposes...
> all <- rbind(cpi_ame, cpi_eu)
Error in match.names(clabs, names(xi)): names do not match previous nam
```

Adding additional rows

`rbind()` requires that all columns in both data frames be named the same:

```
> names(cpi_eu)[1] <- "rank"  # rename for demonstration purposes...
> all <- rbind(cpi_ame, cpi_eu)
Error in match.names(clabs, names(xi)): names do not match previous nam
```

This means you will sometimes need to do some renaming to make things match up properly:

```
> names(cpi_ame)[1] <- "rank"  # rename to match cpi_eu
> all <- rbind(cpi_ame, cpi_eu)
```

Adding additional rows

`rbind()` also requires that all columns be in both data frames:

```
> cpi_eu$CPI2015 <- NULL  # drop for demonstration purposes...
> all <- rbind(cpi_ame, cpi_eu)
Error in rbind(deparse.level, ...): numbers of columns of arguments do
```

Adding additional rows

`rbind()` also requires that all columns be in both data frames:

```
> cpi_eu$CPI2015 <- NULL  # drop for demonstration purposes...
> all <- rbind(cpi_ame, cpi_eu)
Error in rbind(deparse.level, ...): numbers of columns of arguments do
```

To get around this, you can fill in missing columns with NAs:

```
> cpi_eu$CPI2015 <- NA
> all <- rbind(cpi_ame, cpi_eu)
```

Adding additional rows

`rbind()` also requires that all columns be in both data frames:

```
> cpi_eu$CPI2015 <- NULL  # drop for demonstration purposes...
> all <- rbind(cpi_ame, cpi_eu)
Error in rbind(deparse.level, ...): numbers of columns of arguments do
```

To get around this, you can fill in missing columns with NAs:

```
> cpi_eu$CPI2015 <- NA
> all <- rbind(cpi_ame, cpi_eu)
```

Or use the `rbind.fill()` function in the `plyr` library which does this for you:

```
> cpi_eu$CPI2015 <- NULL  # drop for demonstration purposes...
> all <- rbind.fill(cpi_ame, cpi_eu)
```

Adding additional columns

`rbind()` essentially takes two data frames with the same number of columns (but not usually the same number of rows) and sticks them on top of each other.

Similarly, `cbind()` takes two data frames with the same number of rows (but not usually the same number of columns) and sticks them next to each other.

Adding additional columns

`rbind()` essentially takes two data frames with the same number of columns (but not usually the same number of rows) and sticks them on top of each other.

Similarly, `cbind()` takes two data frames with the same number of rows (but not usually the same number of columns) and sticks them next to each other.

It's unusual for data to be structured and stored in such a way that `cbind()` is useful; in particular, there's generally no guarantee that the exact same observations (i.e., rows) are in two separately stored data sets, or that they're sorted the same way.

Instead, for combining columns from different data frames, we usually use `merge()` which handles mismatches between rows by requiring you to specify one or more variables to make this match on (generally some sort of ID)

Adding additional columns

To demonstrate merging, we will load several data frames containing data for US states:

```
> load(paste0(main_dir, "data/us_state_cigarette_data.rdata"),
+     verbose = T)
Loading objects:
  cig_tax
  cig_csmp
  pop
  locs
```

Adding additional columns

```
> summary(cig_tax)
      fips            year         cig_tax
 Min.   : 1.00   Min.   :2010   Min.   :1.180
 1st Qu.:16.00   1st Qu.:2011   1st Qu.:1.630
 Median :29.00   Median :2012   Median :2.370
 Mean   :28.96   Mean   :2012   Mean   :2.512
 3rd Qu.:42.00   3rd Qu.:2013   3rd Qu.:3.010
 Max.   :56.00   Max.   :2014   Max.   :5.360
> summary(cig_csmp)
      fips            year       cig_sales_pc
 Min.   : 1.00   Min.   :2010   Min.   : 15.40
 1st Qu.:16.00   1st Qu.:2011   1st Qu.: 36.20
 Median :29.00   Median :2012   Median : 46.50
 Mean   :28.96   Mean   :2012   Mean   : 50.65
 3rd Qu.:42.00   3rd Qu.:2013   3rd Qu.: 64.10
 Max.   :56.00   Max.   :2014   Max.   :113.00
```

Adding additional columns

```
> summary(pop)
      fips            year           pop
 Min.   : 1.00   Min.   :2011   Min.   :  567768
 1st Qu.:16.00   1st Qu.:2012   1st Qu.: 1644868
 Median :29.00   Median :2013   Median : 4398500
 Mean   :28.96   Mean   :2013   Mean   : 6206177
 3rd Qu.:42.00   3rd Qu.:2014   3rd Qu.: 6862678
 Max.   :56.00   Max.   :2015   Max.   :39144818
> summary(locs)
      fips                state
 Min.   : 1.00   Alabama    : 1
 1st Qu.:16.50   Alaska     : 1
 Median :29.00   Arizona    : 1
 Mean   :28.96   Arkansas   : 1
 3rd Qu.:41.50   California : 1
 Max.   :56.00   Colorado   : 1
                 (Other)    :45
```

Adding additional columns

`merge()` requires that you have one or more variables, present in both data frames, to make a match on. This is specified using the `by` argument:

```
> all <- merge(cig_tax, cig_csmp, by = c("fips", "year"))
> summary(all)
      fips            year         cig_tax       cig_sales_pc
 Min.   : 1.00   Min.   :2010   Min.   :1.180   Min.   : 15.40
 1st Qu.:16.00   1st Qu.:2011   1st Qu.:1.630   1st Qu.: 36.20
 Median :29.00   Median :2012   Median :2.370   Median : 46.50
 Mean   :28.96   Mean   :2012   Mean   :2.512   Mean   : 50.65
 3rd Qu.:42.00   3rd Qu.:2013   3rd Qu.:3.010   3rd Qu.: 64.10
 Max.   :56.00   Max.   :2014   Max.   :5.360   Max.   :113.00
```

It's also possible to use variables that are named differently, using `by.x` and `by.y`.

Adding additional columns

By default, R only keeps rows that match in both data frames.

In the previous example, both data frames contained all the same rows (as defined by `fips` and `year`) so the output of `merge()` has the same number of rows as both input data sets:

```
> dim(cig_tax)
[1] 255   3
> dim(cig_csmp)
[1] 255   3
> dim(all)
[1] 255   4
```

Adding additional columns

By default, R only keeps rows that match in both data frames.

In the previous example, both data frames contained all the same rows (as defined by `fips` and `year`) so the output of `merge()` has the same number of rows as both input data sets:

```
> dim(cig_tax)
[1] 255   3
> dim(cig_csmp)
[1] 255   3
> dim(all)
[1] 255   4
```

This is not always the case:

```
> all <- merge(cig_csmp, pop, by = c("fips", "year"))
> dim(cig_csmp)
[1] 255   3
> dim(pop)
[1] 255   3
> dim(all)
[1] 204   4
```

Adding additional columns

This behavior can be changed using the `all`, `all.x`, or `all.y` arguments:

```
> all <- merge(cig_csmp, pop, by = c("fips", "year"), all = T)
> dim(all)
[1] 306   4
> summary(all)
      fips            year       cig_sales_pc         pop
 Min.   : 1.00   Min.   :2010   Min.   : 15.40   Min.   :  567768
 1st Qu.:16.00   1st Qu.:2011   1st Qu.: 36.20   1st Qu.: 1644868
 Median :29.00   Median :2012   Median : 46.50   Median : 4398500
 Mean   :28.96   Mean   :2012   Mean   : 50.65   Mean   : 6206177
 3rd Qu.:42.00   3rd Qu.:2014   3rd Qu.: 64.10   3rd Qu.: 6862678
 Max.   :56.00   Max.   :2015   Max.   :113.00   Max.   :39144818
                                NA's   :51       NA's   :51
```

Adding additional columns

This behavior can be changed using the `all`, `all.x`, or `all.y` arguments:

```
> all <- merge(cig_csmp, pop, by = c("fips", "year"), all.x = T)
> dim(all)
[1] 255   4
> summary(all)
      fips            year       cig_sales_pc         pop
 Min.   : 1.00   Min.   :2010   Min.   : 15.40   Min.   :  567768
 1st Qu.:16.00   1st Qu.:2011   1st Qu.: 36.20   1st Qu.: 1629301
 Median :29.00   Median :2012   Median : 46.50   Median : 4390584
 Mean   :28.96   Mean   :2012   Mean   : 50.65   Mean   : 6182139
 3rd Qu.:42.00   3rd Qu.:2013   3rd Qu.: 64.10   3rd Qu.: 6841745
 Max.   :56.00   Max.   :2014   Max.   :113.00   Max.   :38792291
                                                 NA's   :51
```

Adding additional columns

This behavior can be changed using the `all`, `all.x`, or `all.y` arguments:

```
> all <- merge(cig_csmp, pop, by = c("fips", "year"), all.y = T)
> dim(all)
[1] 255   4
> summary(all)
      fips            year       cig_sales_pc         pop
 Min.   : 1.00   Min.   :2011   Min.   : 15.40   Min.   :  567768
 1st Qu.:16.00   1st Qu.:2012   1st Qu.: 35.12   1st Qu.: 1644868
 Median :29.00   Median :2013   Median : 45.60   Median : 4398500
 Mean   :28.96   Mean   :2013   Mean   : 49.75   Mean   : 6206177
 3rd Qu.:42.00   3rd Qu.:2014   3rd Qu.: 62.85   3rd Qu.: 6862678
 Max.   :56.00   Max.   :2015   Max.   :107.90   Max.   :39144818
                                NA's   :51
```

Adding additional columns

The examples so far show 1-to-1 merges: the variables specified in `by` uniquely defined the rows in each data frame, so each row in the first data frame matches (at most) one row in the second data frame, and vice versa.

Adding additional columns

A 1-to-many merge is also possible, where one row in the first data frame corresponds to multiple rows in the second data frame:

```
> all <- merge(locs, cig_csmp, by = "fips")
> dim(cig_csmp)
[1] 255   3
> dim(locs)
[1] 51  2
> dim(all)
[1] 255   4
> head(all)
  fips   state year cig_sales_pc
1    1 Alabama 2010         71.5
2    1 Alabama 2011         68.4
3    1 Alabama 2012         67.2
4    1 Alabama 2013         64.6
5    1 Alabama 2014         61.7
6    2  Alaska 2010         43.8
```

(if we swap `locs` and `cig_tax` in the first line above, we'd have a many-to-1 merge, but the effect is the same)

Adding additional columns

Many-to-many merges are also possible – in this case, you get all of the pairwise combinations of rows from each data set (based on the `by` variables specified).

Adding additional columns

Many-to-many merges are also possible – in this case, you get all of the pairwise combinations of rows from each data set (based on the by variables specified).

Many-to-many merges are rarely needed, but it is easy to accidentally do a many-to-many merge if you forget to specifying some of the by variables:

```
> all <- merge(cig_tax, cig_csmp, by = "fips")
> dim(all)
[1] 1275    5
> head(all)
  fips year.x cig_tax year.y cig_sales_pc
1    1   2010   1.435   2010         71.5
2    1   2010   1.435   2011         68.4
3    1   2010   1.435   2012         67.2
4    1   2010   1.435   2013         64.6
5    1   2010   1.435   2014         61.7
6    1   2011   1.435   2010         71.5
```

Saving data

All of the functions we discussed at the beginning for loading data from various file types have corresponding functions for saving data from R to those file types:

.rdata

```
> save(all, file = paste0(main_dir, "output/combined_cig_data.rdata"))
```

.rds

```
> saveRDS(all, file = paste0(main_dir, "output/combined_cig_data.rds"))
```

.csv

```
> write.csv(all, file = paste0(main_dir, "output/combined_cig_data.csv"))
```

.dta

```
> write.dta(all, file = paste0(main_dir, "output/combined_cig_data.dta"))
```

Saving data

A few things to keep in mind when deciding what file type to use for storing data:

- .rdata and .rds are compressed, so the file size is much smaller than most other formats and these files save and load much faster
- .rdata and .rds files can be used for all types of data (not just tabular data) and preserve all of their characteristics
- .rdata and .rds files can NOT be easily loaded in other statistical software such as Stata or Python, so they are not ideal for data being passed between scripts in different languages
- .csv files are convenient since they can be loaded easily into pretty much any program (including Excel)
- But, .csv files are horribly inefficient in terms of file size and can be slow to load

Managing the work space

The `ls()` function can be used to view all of the objects currently in your work space:

```
> ls()
[1] "all"      "cig_csmp" "cig_tax"  "cpi_ame"  "cpi_eu"
[6] "data"     "locs"     "main_dir" "pop"
```

Sometimes this can get cluttered, and we don't want unused objects hanging around taking up memory, so we can remove objects using the `rm()` command:

```
> rm(locs, cig_csmp)
> ls()
[1] "all"      "cig_tax"  "cpi_ame"  "cpi_eu"   "data"
[6] "main_dir" "pop"
```

Managing the work space

You can also use `rm()` to remove all objects, totally clearing the work space:

```
> rm(list = ls())
```

It is standard practice to include this line of code at the top of your scripts so that you're always starting with a clean work space.