

09 - Importing Data

ST 597 | Spring 2017

University of Alabama

11-import.pdf

Contents

1	Getting Started	2
1.1	Required Packages and Data	2
1.2	Using RStudio Import Tools	2
2	Importing Flat Files	2
2.1	readr package	2
2.2	Understanding a Data File	2
2.3	Another Example	4
2.4	Delimited Files	4
2.5	Fixed Width Files	5
2.6	R Functions to know	9
3	Parsing a File	9
3.1	Steps in Data Import of Flat Files	9
3.2	col_types argument	9
3.3	col_names argument	11
3.4	Your Turn: Flat Files	12
3.5	file argument	12
3.6	Saving/Exporting Data Frames	12
3.7	R Functions to know	13
4	Reading Excel Data Tables	13
4.1	readxl package	13
4.2	Example File	13
4.3	read_excel () Options	14
4.4	R Functions to know	15
5	Data in Other Formats	15
5.1	R data formats (.rds, .Rdata)	15
5.2	SAS and SPSS	16
5.3	SQL and Relational Databases	16
5.4	Manual or Clipboard data with scan ()	16
5.5	R Functions to know	17
6	Case Study: APT	18
6.1	The Perfect Job	18

1 Getting Started

1.1 Required Packages and Data

Remember, you may need to `install.packages("pkgname")` before you can load them.

```
library(tidyverse)
library(readxl)
library(stringr)
```

1.2 Using RStudio Import Tools

The recent versions of RStudio provide a GUI to help with file import. Go to `File -> Import Dataset` and choose the type of file: CSV, Excel, SPSS, SAS, or Stata.

Try an example.

Your Turn #1 : Zoo data

1. In RStudio, `File -> Import Dataset -> From CSV` and enter the url for the zoo data <http://www.strategywise.com/Zoo.csv>.
2. Spend a few minutes trying to understand the options.
3. Import the data into R. Notice the code that runs in R.
4. (Optional/Alternative) Open a browser to <http://www.strategywise.com/Zoo.csv> which should prompt you to download the `Zoo.csv` file. Save someplace where you can find it and then direct RStudio to the file.

2 Importing Flat Files

2.1 readr package

The `readr` package will provide our primary functions for importing [flat data files](#), or tabular, into R. That is, these data should naturally be imported into R as a data frame object. The general format is that each row (record or observation) is separated by an end of line (EOL) character and the columns are determined by either: i) delimiters (e.g., comma separated values) or ii) position (e.g., fixed width files).

2.2 Understanding a Data File

To get us started, we will take a simple example. Open your browser to the following url <https://raw.githubusercontent.com/mdporter/ST597/master/data/offers1.csv>.

This is a .csv or comma separated value format. Can you see the role of the commas?

```
name,company,jobtype,location,salary
Tim,GammaRaise Capital,Hedge Fund,San Francisco,87000
Christine,Integral Derivatives,Investment Bank,Chicago,118000
```

```
Lance,Bigup-Side,Startup,"Washington, DC",20000
Bob,Stanguard,Grad School, NYC,20000
Gabrielle,Glitter,Startup, San Francisco,65000
Nick,SocialNET,Startup,Boston,128400
David,InnoTech,Big Software Firm,"Washington, DC",135600
Christine,Irreverent Technologies,Startup, NYC,128400
David,ExcelMacroEconomics,Investment Bank,"Washington, DC",135600
```

Notice a few things:

- The first line is a header: it gives the column names.
- The columns are separated by commas.
- Each observation is on its own line.
- Why is *Washington, DC* in quotes?

We can import the data by rows with the `read_lines()` function:

```
url = "https://raw.githubusercontent.com/mdporter/ST597/master/data/offers1.csv"
(lines = read_lines(url))
#> [1] "name,company,jobtype,location,salary"
#> [2] "Tim,GammaRaise Capital,Hedge Fund,San Francisco,87000"
#> [3] "Christine,Integral Derivatives,Investment Bank,Chicago,118000"
#> [4] "Lance,Bigup-Side,Startup,\"Washington, DC\",20000"
#> [5] "Bob,Stanguard,Grad School, NYC,20000"
#> [6] "Gabrielle,Glitter,Startup, San Francisco,65000"
#> [7] "Nick,SocialNET,Startup,Boston,128400"
#> [8] "David,InnoTech,Big Software Firm,\"Washington, DC\",135600"
#> [9] "Christine,Irreverent Technologies,Startup, NYC,128400"
#> [10] "David,ExcelMacroEconomics,Investment Bank,\"Washington, DC\",135600"
```

This creates a *character vector* showing there are 10 rows. It is clear that each value in a row is separated with a comma (hence, .csv extension). Sometimes the `read_lines()` function is helpful to understand a new dataset.

Question: How does R know that there is a new line after . . . , salary in the first row?

We can actually see the raw file with the `read_file()` function:

```
(file = read_file(url))
#> [1] "name,company,jobtype,location,salary\nTim,GammaRaise Capital,Hedge Fund,San Franci.
```

This function creates a single string of the entire file. Notice that after . . . , salary there is a new line character `\n`. This indicates the start of a new line. When you hit Enter, your program is probably entering a newline character.

2.3 Another Example

Your Turn #2 : Meta data problems

1. Try to load this via the RStudio importer: <https://raw.githubusercontent.com/mdporter/ST597/master/data/offers3.csv>. Something is not correct.
2. Use `read_lines()` to help understand the problem.
3. Fix the problem and load this dataset into R.

2.4 Delimited Files

Delimited files use a delimiter (e.g. comma) to separate the values on a row. While you can always use the function `read_delim()` and set the `delim=` argument, there are some handy shortcuts:

Delimiter	Function	Example of a row
Comma-separated:	<code>read_csv()</code>	1.23,4.56,7.89
Semicolon-separated:	<code>read_csv2()</code>	1.23;4.56;7.89
Tab-separated:	<code>read_tsv()</code>	1.23 4.56 7.89
Pipe-separated:	<code>read_delim(..., delim=" ")</code>	1.23 4.56 7.89

Check out the help for `?read_delim`. Here is a description of some of the arguments (with their default values)

```
read_delim(file,                # path to a file or connection
            delim,              # character used to separate the fields
            col_names = TRUE,   # if `TRUE` will assume the first row is
                                # column names. If the data does not have
                                # column names, then this argument can be
                                # a character vector of column names.
            col_types = NULL,   # specification of the type of data for
                                # for each column
            locale = default_locale(), # set country specific defaults
            na = c("", "NA"),    # character vector of what represents
                                # missing values in the data
            comment = "",        # string used to denote comment lines
            skip = 0,            # number of lines to skip before reading data
            n_max = -1)          # maximum number of rows to read
```

`read_delim()` is looking for a table (data frame), so the data should have rows corresponding to observations and columns corresponding to variables

Remember how quotes were used for "Washington, DC" in the csv file?

Notice that the `quote=` argument is only available with `read_delim()`, so if something other than double quotes (") is used as a quote, then you must use this function instead of `read_{csv, csv2, tsv}`.

Your Turn #3 : Import Delimited data

1. Try to load the file: <https://raw.githubusercontent.com/mdporter/ST597/master/data/offers2a.txt>.
2. Try to load the file: <https://raw.githubusercontent.com/mdporter/ST597/master/data/offers2b.txt>.

2.5 Fixed Width Files

Fixed width files are such that each column is a fixed width and there are no delimiters. Each column starts at a certain distance from the beginning of the line.

An example of a fixed width file is <http://dailydoseofexcel.com/excel/FixedWidthExample2.txt>. Here are the first 29 lines:

```
03/04/2013
Period 01 Thru 03
4:16 pm
Company 200
Page 1
```

Entry	Per.	Post Date	GL Account	Description	Src.	Cflow	Ref.	Post	Debit	Credit	Alloc.
16524	01	10/17/2012	3930621977	TXNPUES	S1	Yes	RHMXWPCP	Yes		5,007.10	No
191675	01	01/14/2013	2368183100	OUNHQEX XUFQONY	S1	No		Yes		43,537.00	Yes
191667	01	01/14/2013	3714468136	GHAKASC QHJXDFM	S1	Yes		Yes	3,172.53		Yes
191673	01	01/14/2013	2632703881	PAHFSAP LUVIKXZ	S1	No		Yes	983.21		No
80495	01	11/21/2012	2766389794	XDZANTV	S1	Yes	TGZGMOXG	Yes		903.78	Yes
80507	01	11/21/2012	4609266335	BWWYEZL	S1	Yes	USUKVMZO	Yes		670.31	No
80509	01	11/21/2012	1092717420	QJYPKVO	S1	No	DNUNTASS	Yes		848.50	Yes
80497	01	11/21/2012	3386366766	SOQLCMU	S1	Yes	BRHUMGJR	Yes		7.31	Yes
191669	01	01/14/2013	5905893739	FYIWNKA QUAFDKD	S1	Yes		Yes	9,167.93		Yes
191671	01	01/14/2013	2749355876	CBMJTLP NGFSEIS	S1	Yes		Yes	746.70		Yes
191674	01	01/14/2013	4530359106	OTAVZGH ZUQFISZ	S1	Yes		No	7,035.74		Yes
244819	01	02/04/2013	4679391677	EGHLQTI ABE	S1	Yes		No		89,947.13	No
96062	01	11/30/2012	5996493062	KTSVTADFF EHEHFMX	S1	Yes	UBNQLRCC	Yes	7.10		Yes
16527	01	10/17/2012	5595769375	ILCVJYC	S1	Yes	HCVZOOMY	Yes		321.19	Yes
191670	01	01/14/2013	1948028853	RPPDCWC UWODNIO	S1	Yes		No	9,293.80		No
191672	01	01/14/2013	4938823703	CTMDXXP HXOXVFF	S1	Yes		No	175.00		Yes
191668	01	01/14/2013	4207018603	DBZZULF QGDZQMD	S1	Yes		Yes	206.26		Yes
ENDING BALANCE PERIOD 01									30,788.27	141,242.32	

- Notice how each column starts and ends at specific positions; the same for each row. Thus, each row is exactly the same length.
- This is different than space or tab (tsv) delimiters which would just add spaces between the column entries. In this case, the starting and stopping position of each column could be different in each row.
- Here is the approach using excel and [ActiveX Data Objects](#) , by the creator of the data.

2.5.1 An R Way

- We will not tackle reading in the entire file now, but rather concentrate on working with the first table to illustrate fixed width files.
- There are two things we need to do:
 1. Find the rows that have the data
 2. Find the positions of the columns

2.5.1.1 Find the rows with the data

- There is some meta data in the first few rows, a space and dashes between the header and data, and the same at the end of the data.
- Use `read_lines()` to see the line numbers

```
url = "http://dailydoseofexcel.com/excel/FixedWidthExample2.txt"
read_lines(url, n_max=29)
```

```
#> [1] "03/04/2013"
#> [2] "Period 01 Thru 03"
#> [3] "4:16 pm"
#> [4] "Company 200"
#> [5] "
#> [6] "
#> [7] " Entry Per. Post Date GL Account Description Srce. Cflow Ref. Post Debit Credit Alloc."
#> [8] "-----"
#> [9] "
#> [10] " 16524 01 10/17/2012 3930621977 TXNPUES S1 Yes RHMXPWPCP Yes 5,007.10 No "
#> [11] "191675 01 01/14/2013 2368183100 OUNHQEX XUFQONY S1 No Yes 43,537.00 Yes "
#> [12] "191667 01 01/14/2013 3714468136 GHAKASC QHJXDFM S1 Yes Yes 3,172.53 Yes "
#> [13] "191673 01 01/14/2013 2632703881 PAHFSAP LUVIKXZ S1 No Yes 983.21 No "
#> [14] " 80495 01 11/21/2012 2766389794 XDZANTV S1 Yes TGZGMOXG Yes 903.78 Yes "
#> [15] " 80507 01 11/21/2012 4609266335 BWWYEZL S1 Yes USUKVMZO Yes 670.31 No "
#> [16] " 80509 01 11/21/2012 1092717420 QJYPKVO S1 No DNUNTASS Yes 848.50 Yes "
#> [17] " 80497 01 11/21/2012 3386366766 SOQLCMU S1 Yes BRHUMGJR Yes 7.31 Yes "
#> [18] "191669 01 01/14/2013 5905893739 FYIWNKA QUAFDKD S1 Yes Yes 9,167.93 Yes "
#> [19] "191671 01 01/14/2013 2749355876 CBMJTLP NGFSEIS S1 Yes Yes 746.70 Yes "
#> [20] "191674 01 01/14/2013 4530359106 OTAVZGH ZUQFISZ S1 Yes No 7,035.74 Yes "
#> [21] "244819 01 02/04/2013 4679391677 EGHQTI ABE S1 Yes No 89,947.13 No "
#> [22] " 96062 01 11/30/2012 5996493062 KTSVTADFF EHEHFMX S1 Yes UBNQLRCC Yes 7.10 Yes "
#> [23] " 16527 01 10/17/2012 5595769375 ILCVJYC S1 Yes HCVZOUY Yes 321.19 Yes "
#> [24] "191670 01 01/14/2013 1948028853 RPPDCWC UWODNIO S1 Yes No 9,293.80 No "
#> [25] "191672 01 01/14/2013 4938823703 CTMDXXP HXOXVFF S1 Yes No 175.00 Yes "
#> [26] "191668 01 01/14/2013 4207018603 DBZZULF QGDZQMD S1 Yes Yes 206.26 Yes "
#> [27] "
#> [28] "
#> [29] "
ENDING BALANCE PERIOD 01 30,788.27 141,242.32"
```

- It looks like:
 - column names (header) on line 7
 - data on lines 10-26
 - we can use `skip=9` and `n_max=17` arguments to get the data

2.5.2 Find the positions of the columns

- I do not know of a simple way to do this. One way is to open the file in a text editor and manually count the spaces.
- One way to do this in R is to use string manipulation tools from the `stringr` package (which is part of tidyverse but not automatically loaded)
- Read in the first few lines (including the header) and create a matrix with one column for each character

```
library(stringr) # need to load stringr package!

#- get first few lines (including the header)
all = read_lines(url)
x = all[c(7, 10, 11)] # only consider lines 7, 10, and 11

#- find the length of each row
str_length(x) # =132

#- use str_split_fixed() function to make matrix
# n=length of row
# pattern='' splits at every character
str_split_fixed(x, pattern='', n=132)
```

```
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14] [,15]
#> [1,] " " "E" "n" "t" "x" "y" " " " " "p" "e" "x" " " " " "p" "o"
#> [2,] " " "1" "6" "5" "2" "4" " " " " "0" "1" " " " " "1" "0" "/"
#> [3,] "1" "9" "1" "6" "7" "5" " " " " "0" "1" " " " " "0" "1" "/"
#>      [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24]
#> [1,] "s" "t" " " "D" "a" "t" "e" " " " "
#> [2,] "1" "7" "/" "2" "0" "1" "2" " " " "
#> [3,] "1" "4" "/" "2" "0" "1" "3" " " " "
```

This shows the line numbers clearly. Now it is a bit easier to see the beginning and end of each field.

- first column (*Entry*) spans 1-6
- second column (*Per.*) spans 9-12
- third column (*Post Date*) spans 13-22
- etc.

RStudio needs a visual aid to help reading in data (Like excel's text to columns). This can be done in Shiny (R code) as an [addin](#). This would be a suitable class project.

2.5.3 Use `read_fwf()` for reading fixed width files

The `readr` function `read_fwf()` is used to read in `fwf` data. There are two options for setting the column positions (`col_positions=`):

- Set the column widths using `fwf_widths()`
- Set the start and stop positions of each column with `fwf_positions()`

Here I will use the `fwf_widths()` option, and setting the widths to span the 132 characters. Trusting `read_fwf()` to take care of the extra white spaces

```
ends = c(8, 12, 22, 35, 63, 68, 73, 82, 86, 104, 124, 132)
widths = diff(c(0,ends)) # difference between ends
read_fwf(url,
  col_positions=fwf_widths(widths),
  skip = 9,
  n_max = 17)
```

```
#> # A tibble: 17 × 12
#>       X1      X2      X3      X4      X5      X6      X7      X8      X9     X10
#>   <int> <chr>   <chr>   <dbl>   <chr> <chr> <chr>   <chr> <chr> <dbl>
#> 1  16524 01 10/17/2012 3.931e+09 TXNPUES S1 Yes RHMXPWPCP Yes NA
#> 2  191675 01 01/14/2013 2.368e+09 OUNHQEX XUFQONY S1 No <NA> Yes NA
#> 3  191667 01 01/14/2013 3.714e+09 GHAKASC QHJXDFM S1 Yes <NA> Yes 3172.5
#> 4  191673 01 01/14/2013 2.633e+09 PAHFSAP LUVIKXZ S1 No <NA> Yes 983.2
#> 5  80495 01 11/21/2012 2.766e+09 XDZANTV S1 Yes TGZGMOXG Yes NA
#> 6  80507 01 11/21/2012 4.609e+09 BWWYEZL S1 Yes USUKVMZO Yes NA
#> 7  80509 01 11/21/2012 1.093e+09 QJYPKVO S1 No DNUNTASS Yes NA
#> 8  80497 01 11/21/2012 3.386e+09 SOQLCMU S1 Yes BRHUMGJR Yes NA
#> 9  191669 01 01/14/2013 5.906e+09 FYIWNKA QUAFDKD S1 Yes <NA> Yes 9167.9
#> 10 191671 01 01/14/2013 2.749e+09 CBMJTLP NGFSEIS S1 Yes <NA> Yes 746.7
#> 11 191674 01 01/14/2013 4.530e+09 OTAVZGH ZUQFISZ S1 Yes <NA> No 7035.7
#> 12 244819 01 02/04/2013 4.679e+09 EGHQTI ABE S1 Yes <NA> No NA
#> 13 96062 01 11/30/2012 5.996e+09 KTSVTADFF EHEHFMX S1 Yes UBNQLRCC Yes 7.1
#> 14 16527 01 10/17/2012 5.596e+09 ILCVJYC S1 Yes HCVZOUMY Yes NA
#> 15 191670 01 01/14/2013 1.948e+09 RPPDCWC UWODNIO S1 Yes <NA> No 9293.8
#> 16 191672 01 01/14/2013 4.939e+09 CTMDXXP HXOXVFF S1 Yes <NA> No 175.0
#> 17 191668 01 01/14/2013 4.207e+09 DBZZULF QGDZQMD S1 Yes <NA> Yes 206.3
#> # ... with 2 more variables: X11 <dbl>, X12 <chr>
```

Note: this reads in the data, but some of the columns are the wrong type (e.g. integers instead of characters, character instead of date). We will use the `col_types=` argument to help read these in correctly.

2.5.3.1 More details

- If all columns are separated by at least one whitespace *and* does not use white space for missing values, try the `read_table()` function. Note: this is not the same as `read_tsv()`, as `read_table()` requires each line to be same length (total width)
- You can let `readr` guess the column positions using `col_positions=fwf_empty(file, skip=)`.
- Both of these only work in special (easy) situations. I expect the usual situation will involve a combination of `read_lines()`, `stringr` functions, and base R functions.
- Here is an example from `?read_fwf`

```
fwf_sample <- system.file("extdata/fwf-sample.txt", package = "readr")
cat(read_lines(fwf_sample))
#> John Smith          WA          418-Y11-4111 Mary Hartford          CA          319-Z19-4341 E

#- You can specify column positions in three ways:
# 1. Guess based on position of empty columns
read_fwf(fwf_sample, fwf_empty(fwf_sample))
#> Parsed with column specification:
#> cols(
#>   X1 = col_character(),
#>   X2 = col_character(),
#>   X3 = col_character(),
#>   X4 = col_character()
#> )
#> # A tibble: 3 × 4
#>       X1      X2      X3      X4
#>   <chr>   <chr> <chr>   <chr>
#> 1 John   Smith   WA  418-Y11-4111
#> 2 Mary   Hartford CA  319-Z19-4341
#> 3 Evan   Nolan   IL  219-532-c301

# 2. A vector of field widths
read_fwf(fwf_sample, fwf_widths(c(2, 5, 3)))
#> Parsed with column specification:
#> cols(
#>   X1 = col_character(),
#>   X2 = col_character(),
#>   X3 = col_character()
#> )
#> # A tibble: 3 × 3
#>       X1      X2      X3
#>   <chr> <chr> <chr>
#> 1 Jo hn Sm ith
#> 2 Ma ry Ha rtf
#> 3 Ev an No lan

# 3. Paired vectors of start and end positions
read_fwf(fwf_sample, fwf_positions(c(1, 4), c(2, 10)))
#> Parsed with column specification:
#> cols(
#>   X1 = col_character(),
#>   X2 = col_character()
```



```
#> )
#> # A tibble: 3 × 2
#>       X1      X2
#>   <chr>  <chr>
#> 1 Jo n Smith
#> 2 Ma y Hartf
#> 3 Ev n Nolan
```

Your Turn #4 : Fixed Width Files

Read this file into R <http://www.cpc.ncep.noaa.gov/data/indices/wksst8110.for>

2.6 R Functions to know

- `read_delim()`
- `read_csv()`
- `read_csv2()`
- `read_tsv()`
- `read_lines()`
- `read_file()`
- `read_fwf()`,
- `fwf_widths()`, `fwf_positions()`, `fwf_empty()`
- `read_table()`

3 Parsing a File

3.1 Steps in Data Import of Flat Files

1. Recognize the file format (csv, fwf, xlsx, etc.)
2. Find the lines of the data component of the file
3. Identify the delimiters or positions of the columns
4. Determine the data type of each column (parse the columns)
5. (Optional) additional preprocessing to clean up the mess
6. Read in the data
 - a. use the correct file format using `read_*()`
 - b. use the correct column parsing using the `col_types=` argument

This section is concerned with 6b, how to set the `col_types=` argument.

3.2 `col_types` argument

- The basic strategy that the `readr` package takes is to initially read in all columns as a character and then convert them using the specifications on the `col_types=` argument.
- If `col_types` is not set (default of `col_types=NULL`), then `readr` uses a heuristic to figure out the data types of your columns:

- it reads the first 1000 rows and uses some (moderately conservative) heuristics to figure out the type of each column.
- This is fast, and fairly robust.
- If readr detects the wrong type of data, you'll get warning messages. readr prints out the first five, and you can access them all with `problems()`.
- If readr does make the correct choice, you can manually set the column types with the `col_types` argument. *OR*, you can use the RStudio import data tool.

3.2.1 Example

Consider the following example <https://raw.githubusercontent.com/mdporter/ST597/master/data/offers4.csv>

```
url4 = "https://raw.githubusercontent.com/mdporter/ST597/master/data/offers4.csv"
read_csv(url4)
```

```
Parsed with column specification:
cols(
  name = col_character(),
  company = col_character(),
  jobtype = col_character(),
  location = col_character(),
  salary = col_character(),
  ID = col_character()
)

# A tibble: 9 × 6
  name      company      jobtype      location      salary      ID
  <chr>      <chr>      <chr>      <chr>      <chr>      <chr>
1 Tim      GammaRaise Capital    Hedge Fund    San Francisco    $87,000    1-1-2016
2 Christine Integral Derivatives Investment Bank    Chicago    $118,000    2-1-2016
3 Lance      Bigup-Side      Startup    Washington, DC    $20,000    3-1-2016
4 Bob      Stanguard      Grad School    NYC    $20,000    4-1-2016
5 Gabrielle Glitter      Startup    San Francisco    $65,000    5-1-2016
6 Nick      SocialNET      Startup    Boston    $128,400    6-1-2016
7 David      InnoTech Big Software Firm    Washington, DC    $135,600    7-1-2016
8 Christine Irreverent Technologies      Startup    NYC    $128,400    8-1-2016
9 David      ExcelMacroEconomics Investment Bank    Washington, DC    $135,600    9-1-2016
```

There are two problems:

1. the salary column should be a number (i.e., remove the \$ and ,)
2. the ID column should be a character vector and not a date object. Check the order of the values in the original csv file!

3.2.2 Manually Setting the column types

There are 4 ways to set the column types

1. Use the RStudio data import tool and select the correct parsing
2. Use `cols()` or `cols_only()` functions
3. Use column type abbreviations
4. Manually convert the columns with e.g., `mutate()`

Here is an example of using the `cols()` function (with abbreviations):

```
read_csv(url4, col_types =
  cols(name="c", company="c", jobtype="c", location="c",
        salary="n",           # number column
        ID = "c") )          # character column
```

```
# A tibble: 9 × 6
  name      company      jobtype      location salary      ID
  <chr>      <chr>      <chr>      <chr>      <dbl>      <chr>
```

```

1      Tim      GammaRaise Capital      Hedge Fund San Francisco 87000 1-1-2016
2 Christine Integral Derivatives      Investment Bank      Chicago 118000 2-1-2016
3 Lance      Bigup-Side      Startup Washington, DC 20000 3-1-2016
4 Bob      Stanguard      Grad School      NYC 20000 4-1-2016
5 Gabrielle Glitter      Startup San Francisco 65000 5-1-2016
6 Nick      SocialNET      Startup Boston 128400 6-1-2016
7 David      InnoTech Big Software Firm Washington, DC 135600 7-1-2016
8 Christine Irreverent Technologies      Startup      NYC 128400 8-1-2016
9 David      ExcelMacroEconomics      Investment Bank Washington, DC 135600 9-1-2016

```

```
# read_csv(url4, col_types="ccccnc") # use column type abbreviations directly
```

The options (with abbreviations) are:

- Special
 - `col_skip()` [`_`, `-`], don't import this column.
 - `col_guess()` [`?`], let readr guess
- Numbers
 - `col_integer()` [`i`], integers.
 - `col_double()` [`d`], doubles.
 - `col_number()` [`n`], finds the first number in the field. A number is defined as a sequence of `-`, `"0-9"`, `decimal_mark` and `grouping_mark`. This is useful for currencies and percentages.
- Dates and Times
 - `col_date(format = "")` [`D`]: Y-m-d dates.
 - `col_datetime(format, tz)`, date times with given format. If the timezone is UTC (the default), this is >20x faster than loading then parsing with `strptime()`.
 - `col_datetime(format = "")` [`T`]: ISO8601 date times
 - `col_time(format)`, times. Returned as number of seconds past midnight.
- Other
 - `col_logical()` [`l`], containing only T, F, TRUE or FALSE.
 - `col_character()` [`c`], everything else.
 - `col_factor(levels, ordered)`, parse a fixed set of known values into a factor

3.2.3 Other Settings

- If you only want to read in certain columns, use `cols_only()` (instead of `cols()`).
 - Or use `col_skip()` or `-`.
- see the `locale=` argument to set default decimal mark, date format, etc
- set the `.default=` argument: `col_types = cols(.default = col_character())`
- The functions `parse_*` can be used directly to convert a vector. These are appropriate for use in `mutate()`

```

read_csv(url4,
  col_types=cols(
    .default=col_character()) # all cols are character vectors
) %>%
mutate(salary = parse_number(salary))

```

- `type_convert()` parses an existing R data frame as if it was reading it in

3.3 col_names argument

The `col_names=` argument has three options:

1. TRUE (the default), which reads column names from the first row of the file
2. FALSE numbers columns sequentially from X1 to Xn
3. A character vector, used as column names. If these don't match up with the columns in the data, you'll get a warning message.

3.4 Your Turn: Flat Files

Your Turn #5 : Flat Files

Read in the data from here <https://raw.githubusercontent.com/mdporter/ST597/master/data/smoke.csv>. The description of the data from: <http://data.princeton.edu/wws509/datasets/#smoking>

- Check the delimiter
- do not read in the first column
- the age column should be an *ordered* factor with levels: `age_levs = c(paste(start, end, sep="-"), "80+")`
- Note any problems with the data

3.5 file argument

The `file` argument can be the path (relative or absolute) to the file or a url.

- Absolute Path
 - `'C:/Users/mdporter/st597/data/sample.csv'`
 - *Note: windows must use forward slash (/) (not default backslash)*
- Relative Path (use `getwd()` to see where you are starting from)
 - `'data/sample.csv'`
 - `'../data/sampleddata/sample.csv'` (use `..` for up directory)
- URL
 - `'http://bama.ua.edu/~mdporter2/st597/data/grades.csv'`
- Also see: `getwd()`, `list.files()`, `file.choose()`

3.6 Saving/Exporting Data Frames

The `readr` functions can write data frames

- `write_csv()`, `write_delim()`
- `write_excel_csv()` is an excel ready csv file

Here is an example of using `file.choose()` to save the path.

```
x = data.frame(x=1:5, y=c('a', 'b', 'c', 'd', 'e'))
write_csv(x, path=file.choose())
```

3.7 R Functions to know

- `cols()`, `col_only()`
- `cols_*`
- `parse_*`
- `type_convert()`
- `getwd()`, `list.files()`, `file.choose()`
- `write_csv()`, `write_delim()`, `write_excel_csv()`

4 Reading Excel Data Tables

4.1 readxl package

```
library(readxl)
```

The `readxl` package lets you load data from both the legacy `.xls` and the modern xml-based `.xlsx` formats into R.

- While `readxl` is part of `tidyverse` it is not loaded automatically, so you must load it with `library(readxl)`
- Note: it is designed to work with *tabular data* stored in a single sheet. While it can get data from different sheets, it does so one sheet at a time.
- Karl Broman has some [good advice](#) for organizing your data in spreadsheets so they can be reused.

There are only two functions in this package. `read_excel()` reads in data as a data frame

```
read_excel(path, sheet = 1, col_names = TRUE, col_types = NULL,  
           na = "", skip = 0)
```

And `excel_sheets()` lists the sheets in an excel spreadsheet.

```
excel_sheets(path)
```

4.2 Example File

The `readxl` package includes some data. The following function will retrieve the path to the data.

```
data_path = system.file("extdata/datasets.xlsx", package = "readxl")
```

We can read in the first sheet (because the default `sheet=1`) with

```
library(readxl)  
read_excel(data_path)  
#> # A tibble: 150 × 5  
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
#>   <dbl>         <dbl>         <dbl>         <dbl>   <chr>  
#> 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
#> 7          4.6          3.4          1.4          0.3 setosa
#> 8          5.0          3.4          1.5          0.2 setosa
#> 9          4.4          2.9          1.4          0.2 setosa
#> 10         4.9          3.1          1.5          0.1 setosa
#> # ... with 140 more rows
```

We can check the name of the sheets:

```
excel_sheets(data_path)
#> [1] "iris"      "mtcars"    "chickwts"  "quakes"
```

OK, let's try the quakes sheet

```
read_excel(data_path, sheet='quakes')
#> # A tibble: 1,000 × 5
#>       lat  long depth  mag stations
#>   <dbl> <dbl> <dbl> <dbl>   <dbl>
#> 1 -20.42 181.6   562  4.8      41
#> 2 -20.62 181.0   650  4.2      15
#> 3 -26.00 184.1    42  5.4      43
#> 4 -17.97 181.7   626  4.1      19
#> 5 -20.42 182.0   649  4.0      11
#> 6 -19.68 184.3   195  4.0      12
#> 7 -11.70 166.1    82  4.8      43
#> 8 -28.11 181.9   194  4.4      15
#> 9 -28.74 181.7   211  4.7      35
#> 10 -17.47 179.6   622  4.3      19
#> # ... with 990 more rows
```

4.3 read_excel() Options

- path path to file (note: does not accept url at the moment)
- col_names if TRUE will assume the first row is column names. If the data does not have column names, then this argument can be a character vector of column names
- col_types can be a character vector of column types (if you know what type of data each column is). If you don't know, it will guess.
 - Note: the options for read_excel() are more limited than the readr package, so may need to use mutate() and parse_*() to get desired results
- na to specify what constitutes a missing value (e.g., 99, NA)
- skip number of rows to skip before reading data. First few rows may be information describing the data.

Your Turn #6 : Excel

You can find an excel file on the course website <https://raw.githubusercontent.com/mdporter/ST597/master/data/offers1.xlsx>

1. Load the data into R
2. Find the average salary.

4.4 R Functions to know

- `read_excel()`
- `excel_sheets()`

5 Data in Other Formats

5.1 R data formats (.rds, .Rdata)

R has its own data formats if you know you will be using data in R exclusively. This is a great option when all your collaborators will use R.

5.1.1 RDS format

You can preserve any *single* R object exactly (e.g., functions, data frames that include factor level information) if you save it in an R format using the `write_rds()` function

```
write_rds(x, path, compress = c("none", "gz", "bz2", "xz"), ...)
```

- Of course, you will only be able to read it with R
- use `.rds` extension in the path.
- use the `compress=` argument to save storage space

Then `read_rds()` will read it back in.

5.1.2 .RData format

Multiple R objects can be saved with the `save()` function.

- List all the R objects to save first, separated by commas
- Extension `.RData` or `.Rda` (they are equivalent)

```
x = "Hello World!"
setosa = filter(iris, Species == 'setosa')
myfunction = median
## save(x, setosa, myfunction, file="data/random.RData")
```

- Objects saved with `save()` can be loaded into the workspace with `load()`

```
rm(x, setosa, myfunction)           # Remove these objects
myfunction = mean                    # change myfunction to mean (from median)
load("data/random.RData")           # Load them back into R
```

- Or use the RStudio Session -> Load Workspace... and look for the file with `.RData` extension.

Be careful, this will overwrite existing R objects with the same name (e.g. `myfunction` will be overwritten back to `median`) **without warning**

5.1.3 Reading R Data from the web

You may need to wrap the url in the R function `url ()` to establish a connection to web data.

Your Turn #7 : Load R Data

```
url1 = 'https://raw.githubusercontent.com/mdporter/ST597/master/data/offers1.rds'  
url2 = 'https://raw.githubusercontent.com/mdporter/ST597/master/data/cars.RData'
```

1. Read in the data <https://raw.githubusercontent.com/mdporter/ST597/master/data/offers1.rds> using `url1`
2. Load the cars `.RData` <https://raw.githubusercontent.com/mdporter/ST597/master/data/cars.RData> using `url2`

5.2 SAS and SPSS

- The [haven package](#) will allow you to read SAS and SPSS data into R.
- Also see the [foreign package](#) for reading and writing data stored by some versions of Epi Info, Minitab, S, SAS, SPSS, Stata, Systat and Weka and for reading and writing some dBase files.

5.3 SQL and Relational Databases

<http://cran.r-project.org/web/packages/dplyr/vignettes/databases.html>

Generally, if your data fits in memory there is no advantage to putting it in a database: it will only be slower and more hassle. The reason you would want to use `dplyr` with a database is because either your data is already in a database (and you do not want to work with static csv files that someone else has dumped out for you), or you have so much data that it does not fit in memory and you have to use a database.

- There is also a discussion of using R to work with databases in [Chapter 3](#) of Spector's book *Data Manipulation with R*

5.4 Manual or Clipboard data with `scan ()`

Data can be entered manually or from the clipboard (i.e., copy data from excel or website) in a couple of ways, but the most flexible is probably with `scan ()`

```
?scan ()
```

`scan ()` will create a vector or list. Consider baseball's 3000 Hit Club data http://en.wikipedia.org/wiki/3,000_hit_club. We want to get the mean career batting average of the players. Select the data from the *Average* column (may need to hold down the `Ctrl` key to select a column) and copy (`Ctrl + c`). Then in R, type the following and hit `Enter`

```
x = scan ()
```

Then paste the data and hit `Enter` again. R should tell you that it *Read 30 items*. Then


```
mean(x)
#> [1] 0.3104
```

The `scan()` function is looking for numeric data by default. If we want to pass in other types of data, we can adjust the `what=` argument. For example, repeat the process to copy the *Team* column

```
team = scan(what=character(), sep="\t")
```

and enter it into R (and another Enter). The `sep=` argument is also needed here. Notice that by default the `scan()` function is looking for a whitespace separator. When we paste from the clipboard, R uses a tab delimiter (`\t` means tab).

There are lots of options for `scan()`; it is a flexible and handy function for quickly getting data into R. Recipe 4.12 from R Cookbook has additional details.

- One way I use `scan()` is to read in the column headers when they are not in the same format as the rest of the data (using `skip=` and `nlines=1` arguments).

Using `scan()` with pasting data from a clipboard does not encourage reproducible research. It is meant to be used for quick, ad hoc analysis. If the data will be further analyzed than saving the data (with details of where and when you obtained the data) or setting up a direct read from source is necessary.

5.5 R Functions to know

- `read_rds()`, `write_rds()`
- `save()`, `load()`
- `url()`
- `scan()`

6 Case Study: APT

6.1 The Perfect Job

APT Analytics company posted an [optimization problem](#) to match employees with employers.

6.1.1 The data

The first step is to examine the data. This is the data from [Sample Input 1](#).

people

Amy | Academic

Bob | Entrepreneur

Charlie | Money Grubber

offers

Amy | MacroHard | Big Software Firm | Seattle

Amy | Stanguard College | Grad School | San Francisco

Amy | Dartboard Modeling | Hedge Fund | NYC

Bob | Bigup-Side | Startup | NYC

Bob | Questionable Tactics | Hedge Fund | San Francisco

Charlie | Cash-Money Inc. | Investment Bank | NYC

Charlie | Arbitrack | Hedge Fund | San Francisco

relationships

Bob | Amy | Dating

Bob | Charlie | Mortal Enemies

This data format is not very nice as it contains three datasets (people, offers, relationships) in a single file.

6.1.2 Scores

There is also the score data from the main webpage. I scraped these and saved them as csv (we will learn how to scrape tables from web later in course). I did some slight cleaning to the column names and values.

```
st597data = 'https://raw.githubusercontent.com/mdporter/ST597/master/data'
```

Jobs

Each type of job has certain benefits and drawbacks along several dimensions:

```
url_job = file.path(st597data, 'scores_job.csv')

(scores_job = read_csv(url_job))
#> # A tibble: 5 × 5
#>   jobtype      Pay Hours Impact Learn
#>   <chr>    <int> <int>   <int> <int>
#> 1 Big Software Firm      6      6      2      8
#> 2 Hedge Fund            8      8      4      6
#> 3 Investment Bank     10     10      3      4
#> 4 Startup              4      8     10      8
#> 5 Grad School          1      4      3     10
```

Personalities

Accordingly, different types of people have different sets of preferences across these job dimensions. These preferences can be thought of as coefficients for the utility offered in each dimension:

```
url_personality = file.path(st597data, 'scores_personality.csv')

(scores_personality = read_csv(url_personality))
#> # A tibble: 4 × 5
#>   personality      Pay Hours Impact Learn
#>   <chr>    <int> <int>   <int> <int>
#> 1 The Money Grubber    10     -1      4      2
#> 2 The Entrepreneur     4     -2     10      8
#> 3 The Slacker          1    -10      2      2
#> 4 The Academic         2     -6      8     10
```

Relationships

Lastly, people don't consider their job choices in a vacuum; their utility derived from a job is a function of both the job itself and the people around them. Since jobs are associated with specific geographies, the geography of the job alone can have a sizable impact on people's happiness:

```
url_relationships = file.path(st597data, 'scores_relationships.csv')

(scores_relationships = read_csv(url_relationships))
#> # A tibble: 4 × 2
#>   relationship      score
#>   <chr>          <chr>
#> 1 Mortal Enemies Cannot be in the same city
#> 2 Friends +20 to each person for same city
#> 3 Dating +50 to each person for same city
#> 4 Married      Must be in the same city
```