Data Wrangling Course

Importing Data

Paths and the Working Directory

The *working directory* is the directory in which R will save files by default. It can be found with `getwd()`. You can change the working directory with `setwd()`. Unless a full path is provided, files are automatically searched for within the working directory.

The readr and readxl Packages

- `readr` is the tidyverse library that includes functions for reading data stored in spreadsheets
  - `read_table`: white space separated values in txt file
  - `read_csv`: comma separated values in a csv
  - `read_csv2`: semi-colon separated values in a csv
  - `read_tsv`: tab delimited values in a tsv
  - `read_delim`: general text file format, must define delimiter (txt)
  - Note that these functions create a *tibble* from the imported data
- `readxl` provides functions to read in data from Microsoft Excel
  - `read_xls`: old-format excel files (xls)
  - `read_xlsx`: new-format excel files (xlsx)
  - `read_excel`: auto-detects format (xls, xlsx)
  - `excel_sheets`: gives the names of the sheets in the excel file, which can be passed to the above functions to determine which sheet is imported
- `read_lines` shows a specified number of lines (argument: `n_max`) of a file – useful if you aren't sure how to import it

Importing Data Using R Base Functions

- R comes with a number of base functions for importing data
  - `read.table`: white space table
  - `read.csv`: comma separated values in a csv
  - `read.delim`: txt file with specified delimiter
  - `read.fwf`: fixed width files
- Note that these functions create *dataframes* rather than *tibbles*
  - use `class` to check the type of the output
  - unless other wise specified (i.e. StringsAsFactors = FALSE), the strings in the data frame will be converted to factors

Downloading Files from the Internet

- To read a file that is located at a url, simply plug in the url as the argument to `read_csv` or its sister functions.
- If you would prefer to download the file, use `download.file(url, "filename.suffix")`.
- Two useful functions during the download and import process are `tempdir` and `tempfile`:
  - `tempdir`: creates a temporary directory name that is very likely to be unique
  - `tempfile`: creates a temporary filename that is very likely to be unique

An example of how these functions might be useful:

```
tmp_filename <- tempfile() #creates an object containing the temporary filename
download.file(url, tmp_filename) #downloads the file from the url to the temporary filename
dat <- read_csv(tmp_filename) #reads the file into R
file.remove(tmp_filename) #removes the temporary file
```

Tidying Data

Tidy Data

- Tidy data (n.): data in which each row represents one observation and the columns represent different variables that we have data on for those observations

Here's an example:

```
path <- system.file("extdata", package = "dslabs") #path on system
filename <- file.path(path, "fertility-two-countries-example.csv") #full path with filename
wide_data <- read.csv(filename) #read in data from path
head(wide_data)
```

```
##       country X1960 X1961 X1962 X1963 X1964 X1965 X1966 X1967 X1968 X1969
## 1     Germany  2.41  2.44  2.47  2.49  2.49  2.48  2.44  2.37  2.28  2.17
## 2 South Korea  6.16  5.99  5.79  5.57  5.36  5.16  4.99  4.85  4.73  4.62
##   X1970 X1971 X1972 X1973 X1974 X1975 X1976 X1977 X1978 X1979 X1980 X1981
## 1  2.04  1.92  1.80  1.70  1.62  1.56  1.53  1.50  1.49  1.48  1.47  1.47
## 2  4.53  4.41  4.27  4.09  3.87  3.62  3.36  3.11  2.88  2.69  2.52  2.38
##   X1982 X1983 X1984 X1985 X1986 X1987 X1988 X1989 X1990 X1991 X1992 X1993
## 1  1.46  1.46  1.46  1.45  1.44  1.43  1.41  1.38  1.36  1.34  1.32  1.31
## 2  2.24  2.11  1.98  1.86  1.75  1.67  1.63  1.61  1.61  1.63  1.65  1.66
##   X1994 X1995 X1996 X1997 X1998 X1999 X2000 X2001 X2002 X2003 X2004 X2005
## 1  1.31  1.31  1.32  1.33  1.34  1.35  1.35  1.35  1.35  1.35  1.35  1.35
## 2  1.65  1.63  1.59  1.54  1.48  1.41  1.35  1.30  1.25  1.22  1.20  1.20
##   X2006 X2007 X2008 X2009 X2010 X2011 X2012 X2013 X2014 X2015
## 1  1.36  1.36  1.37  1.38  1.39  1.40  1.41  1.42  1.43  1.44
## 2  1.20  1.21  1.23  1.25  1.27  1.29  1.30  1.32  1.34  1.36
```

- This *wide* data is different for two import reasons:
  - each row includes several observations
  - one of the variable (the year) is stored in the header

Reshaping Data

The `tidyverse` contains several useful functions for tidying data (which are in the package `tidyr`).

- `gather`: converts wide data into tidy data (default: gather all of the columns)
  - 1st argument: sets the name of the variable that was held in the wide column names (e.g. we could choose `year` for 1960:2015 in the wide_data above)
  - 2nd argument: sets the name of the variable that was held in the column cells (e.g. we could choose `fertility` for the data that was in each of the columns in the wide_data)
  - 3rd argument: specifies which columns should be gathered (e.g. the columns with years in them in the wide_data)
  - Note: the gather function assumes that column names are characters, to detect and change numbers from characters to integers (or doubles), set `convert = TRUE` in gather

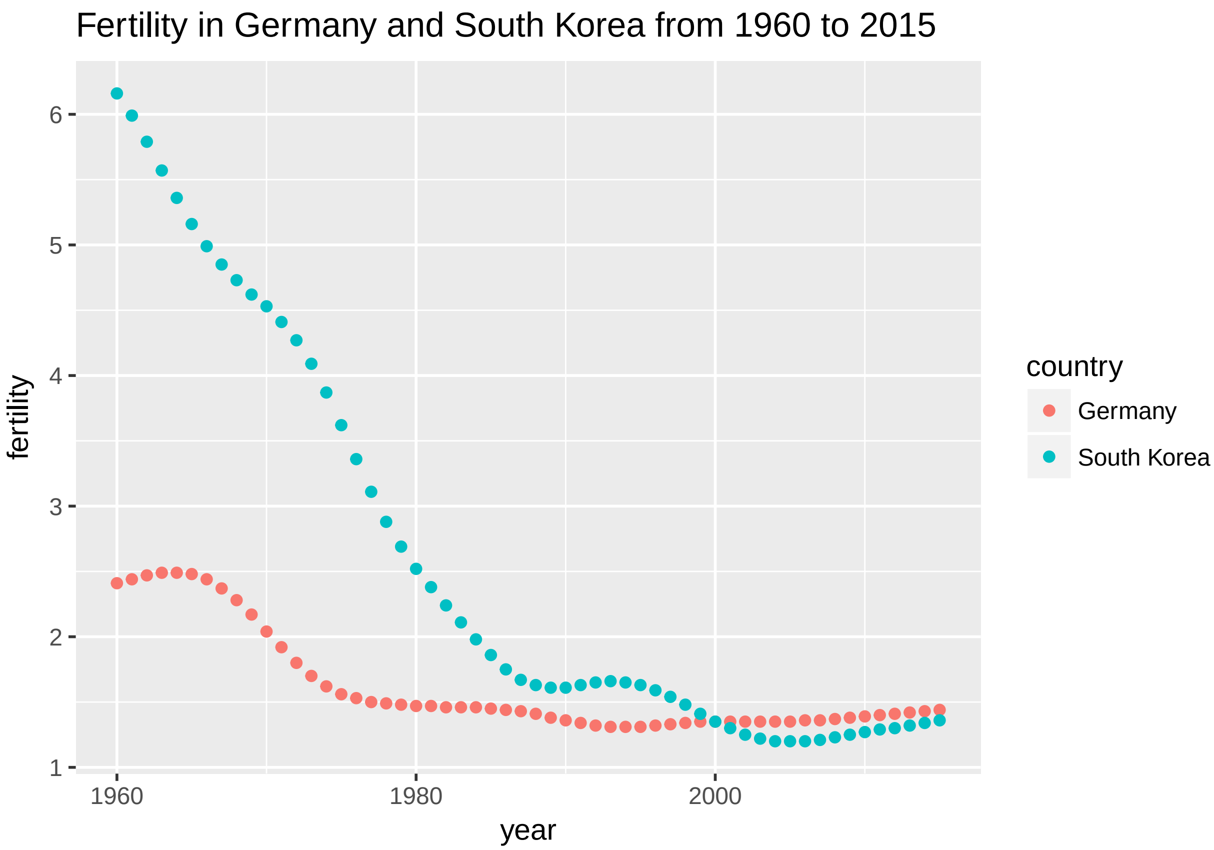
```
new_tidy_data <- wide_data %>%
  gather(key = year, value = fertility, 'X1960':'X2015') # alternatively, could write gather(year, fert
new_tidy_data$year <- as.numeric(substring(new_tidy_data$year, 2,5))
head(new_tidy_data)
```

```
##       country year fertility
## 1     Germany 1960      2.41
## 2 South Korea 1960      6.16
## 3     Germany 1961      2.44
## 4 South Korea 1961      5.99
```

```
## 5     Germany 1962      2.47
## 6 South Korea 1962      5.79
```

Now that the data is tidy, we can use `ggplot` commands to generate a plot of the data.

```
new_tidy_data %>%
  ggplot(aes(year, fertility, color = country)) +
  geom_point() +
  ggtitle("Fertility in Germany and South Korea from 1960 to 2015")
```



- `spread`: performs the inverse function of gather, which is sometimes useful as an intermediate step in tidying data
  - 1st argument: which variable should be used as the column names
  - 2nd argument: which variables should be used to fill out the cells

```
new_wide_data <- new_tidy_data %>% spread(year, fertility)
select(new_wide_data, country, 2:9)
```

```
##       country 1960 1961 1962 1963 1964 1965 1966 1967
## 1     Germany 2.41 2.44 2.47 2.49 2.49 2.48 2.44 2.37
## 2 South Korea 6.16 5.99 5.79 5.57 5.36 5.16 4.99 4.85
```

Separate and Unite

Let's take a more complicated example.

```
path <- system.file("extdata", package = "dslabs") #path on system
filename <- file.path(path, "life-expectancy-and-fertility-two-countries-example.csv") #full path with
raw_dat <- read_csv(filename)
```

```
## Parsed with column specification:
```

```
## cols(
##   .default = col_double(),
##   country = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

```
select(raw_dat, 1:5)
```

```
## # A tibble: 2 x 5
##   country     `1960_fertility` `1960_life_expectancy` `1961_fertility`
##   <chr>                  <dbl>                  <dbl>            <dbl>
## 1 Germany                 2.41                   69.3             2.44
## 2 South Korea             6.16                   53.0             5.99
## # ... with 1 more variable: `1961_life_expectancy` <dbl>
```

Note the way fertility and life expectancy are stored in the column names.

First, we gather the data excluding country. We'll use the default for the key and value names, since they aren't tidy variables yet.

```
dat <- raw_dat %>% gather(key, value, -country)
head(dat)
```

```
## # A tibble: 6 x 3
##   country     key                  value
##   <chr>       <chr>                <dbl>
## 1 Germany     1960_fertility        2.41
## 2 South Korea 1960_fertility        6.16
## 3 Germany     1960_life_expectancy 69.3
## 4 South Korea 1960_life_expectancy 53.0
## 5 Germany     1961_fertility        2.44
## 6 South Korea 1961_fertility        5.99
```

The `readr` package includes functions to deal with the common problem of multiple variables encoded in a column. * `separate`: separates the offending column into two are more variables + 1st argument: the column to be separated + 2nd argument: names to be used for the new columns + 3rd argument: character that separates the variables + Note: the default separate is `_`, so the `sep` argument could have been dropped below

```
dat %>% separate(key, c("year", "variable_name"), sep = "_")
```

```
## Warning: Expected 2 pieces. Additional pieces discarded in 112 rows [3, 4,
## 7, 8, 11, 12, 15, 16, 19, 20, 23, 24, 27, 28, 31, 32, 35, 36, 39, 40, ...].
```

```
## # A tibble: 224 x 4
##    country     year  variable_name value
##    <chr>       <chr> <chr>         <dbl>
##  1 Germany     1960  fertility      2.41
##  2 South Korea 1960  fertility      6.16
##  3 Germany     1960  life          69.3
##  4 South Korea 1960  life          53.0
##  5 Germany     1961  fertility      2.44
##  6 South Korea 1961  fertility      5.99
##  7 Germany     1961  life          69.8
##  8 South Korea 1961  life          53.8
##  9 Germany     1962  fertility      2.47
## 10 South Korea 1962  fertility      5.79
## # ... with 214 more rows
```

Note that this code produces a warning due to the underscore within `life_expectancy`. (Life_expectancy has been truncated to life in the separated data.) One way to address this is to go back and add a third variable name and specify that the missing values be filled in on the right.

```
dat %>% separate(key, c("year", "variable_name", "second_variable_name"), fill = "right", sep = "_") %>%
```

```
## # A tibble: 6 x 5
##   country     year  variable_name second_variable_name value
##   <chr>       <chr> <chr>         <chr>                <dbl>
## 1 Germany     1960  fertility     <NA>                  2.41
## 2 South Korea 1960  fertility     <NA>                  6.16
## 3 Germany     1960  life          expectancy           69.3
## 4 South Korea 1960  life          expectancy           53.0
## 5 Germany     1961  fertility     <NA>                  2.44
## 6 South Korea 1961  fertility     <NA>                  5.99
```

Even better, we could specify that extra columns be *merged*.

```
dat %>% separate(key, c("year", "variable_name"), extra = "merge", sep = "_") %>% head()
```

```
## # A tibble: 6 x 4
##   country     year  variable_name   value
##   <chr>       <chr> <chr>           <dbl>
## 1 Germany     1960  fertility        2.41
## 2 South Korea 1960  fertility        6.16
## 3 Germany     1960  life_expectancy 69.3
## 4 South Korea 1960  life_expectancy 53.0
## 5 Germany     1961  fertility        2.44
## 6 South Korea 1961  fertility        5.99
```

One more step to tidy data: we need `fertility` and `life_expectancy` as columns. For this, we can use `spread`.

```
dat %>% separate(key, c("year", "variable_name"), extra = "merge", sep = "_") %>%
  spread(variable_name, value)
```

```
## # A tibble: 112 x 4
##    country year  fertility life_expectancy
##    <chr>   <chr>     <dbl>           <dbl>
##  1 Germany 1960       2.41            69.3
##  2 Germany 1961       2.44            69.8
##  3 Germany 1962       2.47            70.0
##  4 Germany 1963       2.49            70.1
##  5 Germany 1964       2.49            70.7
##  6 Germany 1965       2.48            70.6
##  7 Germany 1966       2.44            70.8
##  8 Germany 1967       2.37            71.0
##  9 Germany 1968       2.28            70.6
## 10 Germany 1969       2.17            70.5
## # ... with 102 more rows
```

- `unite` works in the opposite fashion to `separate`. You can see how it functions in the example below, where we separate the columns above and then reunite them with `unite`. This is less efficient, but demonstrative of how `unite` functions.

```
dat %>% separate(key, c("year", "first_variable_name", "second_variable_name"), fill = "right", sep = "
```

```
## # A tibble: 224 x 4
```

```
##    country     year  variabile_name  value
##    <chr>       <chr> <chr>           <dbl>
##  1 Germany     1960  fertility_NA     2.41
##  2 South Korea 1960  fertility_NA     6.16
##  3 Germany     1960  life_expectancy 69.3
##  4 South Korea 1960  life_expectancy 53.0
##  5 Germany     1961  fertility_NA     2.44
##  6 South Korea 1961  fertility_NA     5.99
##  7 Germany     1961  life_expectancy 69.8
##  8 South Korea 1961  life_expectancy 53.8
##  9 Germany     1962  fertility_NA     2.47
## 10 South Korea 1962  fertility_NA     5.79
## # ... with 214 more rows
```

To finish up this example, we would then spread the columns and rename fertility with this code:

```
dat %>% separate(key, c("year", "first_variable_name", "second_variable_name"), fill = "right", sep = "
  unite(variable_name, first_variable_name, second_variable_name, sep = "_") %>%
  spread(variable_name, value) %>%
  rename(fertility = fertility_NA)
```

```
## # A tibble: 112 x 4
##    country year  fertility life_expectancy
##    <chr>   <chr>     <dbl>           <dbl>
##  1 Germany 1960       2.41            69.3
##  2 Germany 1961       2.44            69.8
##  3 Germany 1962       2.47            70.0
##  4 Germany 1963       2.49            70.1
##  5 Germany 1964       2.49            70.7
##  6 Germany 1965       2.48            70.6
##  7 Germany 1966       2.44            70.8
##  8 Germany 1967       2.37            71.0
##  9 Germany 1968       2.28            70.6
## 10 Germany 1969       2.17            70.5
## # ... with 102 more rows
```

Combining Tables

Joining tables is often useful when multiple tables contain information about the same thing (e.g. a table with state populations and one with state electoral vote counts).

- `left_join` combines information from two tables, matching them based on a column to ensure that each row is constructed correctly.

Joining the `murders` and `results_us_election_2016` data directly wouldn't work since the state columns are not identical (i.e. some of the states are in a different order).

```
data(murders)
data(polls_us_election_2016)
identical(results_us_election_2016, murders$state)
```

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

Instead, we join with `left_join`, which ensures that the tables are joined properly.

```
tab <- left_join(murders, results_us_election_2016, by = "state")
head(tab)
```

```
##        state abb region population total electoral_votes clinton trump
```

```
## 1    Alabama  AL  South    4779736   135              9    34.4  62.1
## 2     Alaska  AK   West     710231    19              3    36.6  51.3
## 3    Arizona  AZ   West    6392017   232             11    45.1  48.7
## 4   Arkansas  AR  South    2915918    93              6    33.7  60.6
## 5 California  CA   West   37253956  1257             55    61.7  31.6
## 6   Colorado  CO   West    5029196    65              9    48.2  43.3
##   others
## 1    3.6
## 2   12.2
## 3    6.2
## 4    5.8
## 5    6.7
## 6    8.6
```

In practice, it is not always the case that each row in one table has a matching row in the other. Here's an example demonstrating how to deal with this problem.

```
tab1 <- slice(murders, (1:6)) %>% select(state, population)
```

```
## Warning: package 'bindrcpp' was built under R version 3.2.5
```

```
tab2 <- slice(results_us_election_2016, c(1:3, 5, 7:8)) %>% select(state, electoral_votes)
```

Joining with `left_join`, we get NAs were there aren't electoral votes available.

```
left_join(tab1, tab2)
```

```
## Joining, by = "state"
```

```
## # A tibble: 6 x 3
##   state      population electoral_votes
##   <chr>           <dbl>           <int>
## 1 Alabama       4779736               9
## 2 Alaska         710231               3
## 3 Arizona       6392017              11
## 4 Arkansas      2915918              NA
## 5 California   37253956              55
## 6 Colorado      5029196              NA
```

```
#could also write tab1 %>% left_join(tab2)
```

Joining with `right_join` keeps the all of the columns from `tab2` and produces NAs where populations are not available.

```
tab1 %>% right_join(tab2)
```

```
## Joining, by = "state"
```

```
## # A tibble: 6 x 3
##   state       population electoral_votes
##   <chr>            <dbl>           <int>
## 1 Alabama        4779736               9
## 2 Alaska          710231               3
## 3 Arizona        6392017              11
## 4 California    37253956              55
## 5 Connecticut         NA               7
## 6 Delaware            NA               3
```

Joining with `inner_join` only keeps the rows where there was data from each table.

```r
tab1 %>% inner_join(tab2)
```

```
## Joining, by = "state"

## # A tibble: 4 x 3
##   state      population electoral_votes
##   <chr>           <dbl>           <int>
## 1 Alabama       4779736               9
## 2 Alaska         710231               3
## 3 Arizona       6392017              11
## 4 California   37253956              55
```

Joining with `full_join` keeps all of the rows, generating NAs were data is not available.

```r
tab1 %>% full_join(tab2)
```

```
## Joining, by = "state"

## # A tibble: 8 x 3
##   state       population electoral_votes
##   <chr>            <dbl>           <int>
## 1 Alabama        4779736               9
## 2 Alaska          710231               3
## 3 Arizona        6392017              11
## 4 Arkansas       2915918              NA
## 5 California    37253956              55
## 6 Colorado       5029196              NA
## 7 Connecticut         NA               7
## 8 Delaware            NA               3
```

Joining by `semi_join` keeps only the parts of the first table that are present in the second *without adding anything from the second table.*

```r
tab1 %>% semi_join(tab2)
```

```
## Joining, by = "state"

## # A tibble: 4 x 2
##   state      population
##   <chr>           <dbl>
## 1 Alabama       4779736
## 2 Alaska         710231
## 3 Arizona       6392017
## 4 California   37253956
```

Joining by `anti_join` keeps only the parts of the first table that are *not* available in the second *without adding anything from the second table.*

```r
tab1 %>% anti_join(tab2)
```

```
## Joining, by = "state"

## # A tibble: 2 x 2
##   state    population
##   <chr>         <dbl>
## 1 Arkansas    2915918
## 2 Colorado    5029196
```

Binding

Binding functions combine tables regardless of variables. If the dimensions don't match, you will obtain an error.

- `bind_cols` binds two objects by putting the columns together in a table (e.g. `bind_cols(a = 1:3, b = 4:6)`)
  - this function usually creates a tibble, but will create a dataframe if fed two dataframes
- `cbind` also binds objects into columns, but creates matrices or dataframes rather than tibbles
- `bind_rows` binds rows as above (to a tibble)
- `rbind` binds rows as above (to a matrix or dataframe)

Set Operators

The following functions can be applied to vectors to produce various subsets. Note that if the `tidyverse` (specifically, `dplr`) has been loaded, these functions can also be used on dataframes.

- `intersect` takes the intersection of two vectors (or tables) finding which elements (or rows) are present in both
- `union` takes the union of two vectors (or tables) showing all of the unique elements (or rows)
- `setdiff` shows which elements (or rows) are different between two vectors (or tables)
- `setequal` tells us if two sets are the same regardless of order
- with dplyr loaded, `setequal` will return the columns and rows that are different between two tables

Web Scraping

The `tidyverse` contains a package called `rvest` that contains functions for reading html pages into R.

```
library(rvest)
```

```
## Warning: package 'rvest' was built under R version 3.2.5

## Loading required package: xml2

##
## Attaching package: 'rvest'

## The following object is masked from 'package:purrr':
##
##     pluck

## The following object is masked from 'package:readr':
##
##     guess_encoding
```

```
url <- "https://en.wikipedia.org/wiki/Gun_violence_in_the_United_States_by_state"
h <- read_html(url)
```

- `html_nodes` extracts all nodes (classes between <>) of a given type

- `html_node` extracts the first node of a given type

```
tab <- h %>% html_nodes("table")
tab <- tab[[2]]
tab <- tab %>% html_table
tab <- tab %>% setNames(c("state", "population", "total", "murders", "gun_murders", "gun_ownership", "t
```

CSS Selectors

CSS, or cascading style sheets, are what make webpages look nice. CSS relies on selectors to define recurring elements in a page such as headers and font size. There are many types and finding them can be complicated. Rafa recommends downloading `SelectorGadget` to find the correct ones in a webpage.

Here's an example using `html_nodes` to get a guacamole recipe off of the food network's website:

```r
#note the complexity of the selectors
h <- read_html("http://www.foodnetwork.com/recipes/alton-brown/guacamole-recipe-1940609")
recipe <- h %>% html_node(".o-AssetTitle__a-HeadlineText") %>% html_text()
prep_time <- h %>% html_node(".o-RecipeInfo__a-Description--Total") %>% html_text()
ingredients <- h %>% html_nodes(".o-Ingredients__a-ListItemText") %>% html_text()

#creating a list
guacamole <- list(recipe, prep_time, ingredients)
guacamole

#creating a function to make scraping recipies easy!
get_recipe <- function(url){
    h <- read_html(url)
    recipe <- h %>% html_node(".o-AssetTitle__a-HeadlineText") %>% html_text()
    prep_time <- h %>% html_node(".o-RecipeInfo__a-Description--Total") %>% html_text()
    ingredients <- h %>% html_nodes(".o-Ingredients__a-ListItemText") %>% html_text()
    return(list(recipe = recipe, prep_time = prep_time, ingredients = ingredients))
}
#use the function on any other food network recipe
get_recipe("http://www.foodnetwork.com/recipes/food-network-kitchen/pancakes-recipe-1913844")
```

Note also that there are other functions such as `html_form`, `set_values`, and `submit_form` which allow you to query a webpage from R.

String Processing

Parsing Strings

It's very common to encounter numbers that R reads in as characters because they contain commas or other characters that improve readability (e.g. 4,542,543 rather than 4542543). We need to convert those into numbers to do operations on them in R.

- `parse_number` removes commas from numbers and converts them into numeric form

Single and Double Quotes: Escaping Strings

If you run into a situation where strings are not easily added since they include quotes, you can "escape" them with a backslash. + e.g. To create an object that contains the string `5'10''`, we could use `\`in one of two ways: `'5\'10"'` or `"5'10\""` + Note: R will otherwise recognize the quotes as missing their pair, and will expect additional input

The stringr package

The `stringr` package contains a variety of string modification functions which all begin with `str_` so they are easily found by hitting tab after writing the prefix. They also all work well with the pipe, since the string is always the first argument.

- `str_replace` replaces elements of strings based on a pattern
- `str_detect` returns a logical whether the string fits the given pattern
- `str_subset` returns a subset of the data based on the given pattern
- `str_match` returns the data for which there was a match and the groupings in separate columns; non-matches are returned as NAs
- `str_trim` removes spaces at the start or end of a string

- `str_to-lower` makes a letters lowercase
- `str_split` separates strings with separators (e.g. commas) into lists

Case Study 1: US Murders Data

Let's use the raw murders data we extracted from Wikipedia earlier. We can use `str_detect` to see which columns contain commas in our dataset.

```
murders_raw <- tab
commas <- function(x) any(str_detect(x, ","))
murders_raw %>% summarize_all(funs(commas))
```

```
##   state population total murders gun_murders gun_ownership total_rate
## 1 FALSE       TRUE  TRUE    TRUE        TRUE         FALSE      FALSE
##   murder_rate gun_murder_rate
## 1       FALSE           FALSE
```

We can then use `str_replace_all` to remove those commas.

```
test_1 <- str_replace_all(murders_raw$population, ",", "")
test_1
```

```
##  [1] "4853875"  "737709"   "6817565"  "2977853"  "38993940" "5448819"
##  [7] "3584730"  "944076"   "670377"   "20244914" "10199398" "1425157"
## [13] "1652828"  "12839047" "6612768"  "3121997"  "2906721"  "4424611"
## [19] "4668960"  "1329453"  "5994983"  "6784240"  "9917715"  "5482435"
## [25] "2989390"  "6076204"  "1032073"  "1893765"  "2883758"  "1330111"
## [31] "8935421"  "2080328"  "19747183" "10035186" "756835"   "11605090"
## [37] "3907414"  "4024634"  "12791904" "1055607"  "4894834"  "857919"
## [43] "6595056"  "27429639" "2990632"  "626088"   "8367587"  "7160290"
## [49] "1841053"  "5767891"  "586107"
```

Commas in numbers are such a common occurence that `parse_number` is specifically designed to address them. We could write:

```
murders_new <- murders_raw %>% mutate_at(2:3, parse_number)
murders_new %>% head
```

```
##        state population total murders gun_murders gun_ownership total_rate
## 1    Alabama    4853875   348    3[a]        3[a]          48.9        7.2
## 2     Alaska     737709    59      57          39          61.7        8.0
## 3    Arizona    6817565   309     278         171          32.3        4.5
## 4   Arkansas    2977853   181     164         110          57.9        6.1
## 5 California   38993940  1861   1,861       1,275          20.1        4.8
## 6   Colorado    5448819   176     176         115          34.3        3.2
##   murder_rate gun_murder_rate
## 1      0.1[a]          0.1[a]
## 2         7.7             5.3
## 3         4.1             2.5
## 4         5.5             3.7
## 5         4.8             3.3
## 6         3.2             2.1
```

Case Study 2: Reported Heights

The heights data from the dslabs package had some inconsistent entries in its raw form. Let's make them consistent. We see that `as.numeric` introduces NAs because some of the entries are not numbers.

```
data("reported_heights")
heights <- as.numeric(reported_heights$height)
```

```
## Warning: NAs introduced by coercion
```

We can use `filter` to find which entries are being coerced to NAs.

```r
reported_heights %>% mutate(new_height = as.numeric(height)) %>%
  filter(is.na(new_height)) %>% head(10)
```

```
## Warning in eval(substitute(expr), envir, enclos): NAs introduced by
## coercion

##              time_stamp    sex                 height new_height
## 1  2014-09-02 15:16:28   Male                  5' 4"         NA
## 2  2014-09-02 15:16:37 Female                  165cm         NA
## 3  2014-09-02 15:16:52   Male                    5'7         NA
## 4  2014-09-02 15:16:56   Male                  >9000         NA
## 5  2014-09-02 15:16:56   Male                   5'7"         NA
## 6  2014-09-02 15:17:09 Female                   5'3"         NA
## 7  2014-09-02 15:18:00   Male 5 feet and 8.11 inches         NA
## 8  2014-09-02 15:19:48   Male                   5'11         NA
## 9  2014-09-04 00:46:45   Male                  5'9''         NA
## 10 2014-09-04 10:29:44   Male                 5'10''         NA
```

Let's create a function to identify all of the problem entries (finding all NAs and anything outside realistic heights).

```r
not_inches <- function(x, smallest = 50, tallest = 84){
  inches <-suppressWarnings(as.numeric(x))
  ind <- is.na(inches) | inches < smallest | inches > tallest
  ind
}
```

Let's apply this function to find our problem entries.

```r
problems <- reported_heights %>%
  filter(not_inches(height)) %>%
  .$height
length(problems)
```

```
## [1] 292
```

Looking at the problem examples, it becomes clear that there are entries formatted with single and double quotes to denote feet and inches, entries in the same format but using periods or commas, and entries in centimeteres rather than inches.

Regex

Regular expressions (regex) are patterns that can be used as arguments to functions to search for strings meeting certain criteria.

- `|` denotes "or". For example, we can detect whether a string contains "cm" or "inches" with 'str_detect(string, "cm|inches").
- `\\d` represents any digit (0:9). The backslash distinguishes it from the character. In R, we have to "escape" the backslash, so we'll use two backslashes.
- `\\s` represents a space.
- `\\.` represents a period, since it is a special character and needs to be escaped. Otherwise, a period (`.`) means any character except a line break.

Character Classes, Anchors, and Quantifiers

- `[ ]` are character brackets which will cause functions to use the presence of any of the *individual* characters in the brackets as search criteria.
  - Use a `-` to input a range (e.g `[4-7]` searches for any instances of 4, 5, 6, or 7).
  - The same works for the alphabet (e.g. `[a-z]` searches for all of the lowercase letters).
  - All letters would be `[a-zA-Z]`.
- Anchors the beginning and end of patterns. `^` is the beginning of a string, and `$` is the end.
  - For example, `^//d$` only detects strings with a single digit.
  - You can use these alone to specify the first or last character of string (e.g. `[A-Z]$` specifies that the last character is a capital letter).
- Add a quantifier, `{}`, to allow for a range of possibilities (e.g. `\\d{1,2}` looks for 1 or 2 digits).
  - The `*` means zero or more instances, and can be used as a quantifier.
  - The `?` means none or once.
  - The `+` means one or more times.

Here's a table to make the quantifiers a bit more clear.

```
yes <- c("AB", "A1B", "A11B", "A111B", "A1111B")
data.frame(string = c("AB", "A1B", "A11B", "A111B", "A1111B"),
           none_or_more = str_detect(yes, "A1*B"),
           none_or_once = str_detect(yes, "A1?B"),
           once_or_more = str_detect(yes, "A1+B"))
```

```
##   string none_or_more none_or_once once_or_more
## 1     AB         TRUE         TRUE        FALSE
## 2    A1B         TRUE         TRUE         TRUE
## 3   A11B         TRUE        FALSE         TRUE
## 4  A111B         TRUE        FALSE         TRUE
## 5 A1111B         TRUE        FALSE         TRUE
```

Combining these rules for our case study, we can create a pattern that will search for one of the common formats in which heights were inputted above.

```
pattern <- "^[4-7]'\\d{1,2}\"$"
```

In the pattern above, the carrot denots the start of the string, followed by any number from 4 to 7, then any 1 or 2 digits, then a double quote, followed by the end of the string.

Search and Replace with Regex

Let's use search and replace to make editing the incorrect submissions easier. Let's adjust the pattern to no longer consider the double quote at the end.

```
pattern <- "^[4-7]'\\d{1,2}$"
problems %>%
  str_replace("feet|ft|foot", "'") %>%
  str_replace("inches|in|''|\"", "") %>%
  str_detect(pattern) %>%
  sum
```

```
## [1] 48
```

Our pattern is picking up a lot more, but not all of the problem entries. Some have spaces between the feet and inches numbers, which R interprets as not fitting our pattern. Let's adjust with `\\s` with a quantifier to include these.

```
pattern <- "^[4-7]\\s*'\\s*\\d{1,2}$"
problems %>%
  str_replace("feet|ft|foot", "'") %>%
```

```r
  str_replace("inches|in|''|\"", "") %>%
  str_detect(pattern) %>%
  sum
```

```
## [1] 53
```

Groups with Regex

- Groups allow tools to identify specific parts of a pattern so that they can be extracted.

Here's an example, wherein we need to find a pattern that catches heights entered with periods or commas without mistakenly changing correctly formatted numbers.

```r
pattern_without_groups <- "^[4-7],\\d*$"
pattern_with_groups <- "^([4-7]),(\\d*$)" #encapsulate the parts we want to keep
```

Note that this doesn't affect the identification process. It only helps us extract pieces we want.

```r
yes <- c("5,9", "5,11", "6,", "6,1")
no <- c("5'9", ",", "2,8", "6.1.1")
s <- c(yes, no)
str_detect(s, pattern_without_groups)
```

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

```r
str_detect(s, pattern_with_groups)
```

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

With `str_match`, we can separate out the two digits from the original code.

```r
str_match(s, pattern_with_groups)
```

```
##      [,1]   [,2] [,3]
## [1,] "5,9"  "5"  "9"
## [2,] "5,11" "5"  "11"
## [3,] "6,"   "6"  ""
## [4,] "6,1"  "6"  "1"
## [5,] NA     NA   NA
## [6,] NA     NA   NA
## [7,] NA     NA   NA
## [8,] NA     NA   NA
```

In contrast, `str_extract` would have produced a vector of elements that matched the criteria.

You can refer to the extracted value in regex when searching and replacing.

- `\\i` refers to the ith group, so `\\1` is the first group and `\\2` is the second.

Here's how we can use it:

```r
str_replace(s, pattern_with_groups, "\\1'\\2")
```

```
## [1] "5'9"   "5'11"  "6'"    "6'1"   "5'9"   ","     "2,8"   "6.1.1"
```

Putting it all together:

```r
patterns_with_groups <- "^([4-7])\\s*[,\\.\\s+]\\s*(\\d*)$"
```

The indicates the beginning of the string, the range 4:7, none or more spaces, the next symbol is a comma, period, or one or more spaces, none or more spaces, any digit, and the end of the string.

Now we can search and replace:

```
str_subset(problems, patterns_with_groups) %>%
  str_replace(patterns_with_groups, "\\1'\\2") %>% head()
```

```
## [1] "5'3"  "5'25" "5'5"  "6'5"  "5'8"  "5'6"
```

We're almost there! We just need to deal with the 25in.

Let's create a function that captures all entities that cannot be converted into numbers, excluding those entered in cm (even though we'd have to fix those later).

```
not_inches_or_cm <- function(x, smallest = 50, tallest = 84){
  inches <- suppressWarnings(as.numeric(x))
  ind <- !is.na(inches) &
    ((inches >= smallest & inches <= tallest) |
       (inches/2.54 >= smallest & inches/2.54 <= tallest))
  !ind
}
problems <- reported_heights %>%
  filter(not_inches_or_cm(height)) %>% .$height
length(problems)
```

```
## [1] 200
```

Let's convert with our pattern above, and see how many we fix.

```
converted <- problems %>%
  str_replace("feet|foot|ft", "'") %>% #convert feet symbols to '
  str_replace("inches|in|''|\"", "") %>% #remove inch symbols
  str_replace("^([4-7])\\s*[,\\.\\s+]\\s*(\\d*)$", "\\1'\\2") # change the format
pattern <- "^[4-7]\\s*'\\s*\\d{1,2}$"
index <- str_detect(converted, pattern) #logical: do they match the pattern we wanted?
mean(index)
```

```
## [1] 0.615
```

After trying to deal with nearly every case, you could end up creating the following functions and plugging in the problem list:

```
words_to_numbers <- function(s){
  str_to_lower(s) %>%
    str_replace_all("zero", "0") %>%
    str_replace_all("one", "1") %>%
    str_replace_all("two", "2") %>%
    str_replace_all("three", "3") %>%
    str_replace_all("four", "4") %>%
    str_replace_all("five", "5") %>%
    str_replace_all("six", "6") %>%
    str_replace_all("seven", "7") %>%
    str_replace_all("eight", "8") %>%
    str_replace_all("nine", "9") %>%
    str_replace_all("ten", "10") %>%
    str_replace_all("eleven", "11")
}
convert_format <- function(s){
  s %>%
    str_replace("feet|foot|ft", "'") %>% #convert feet symbols to '
    str_replace_all("inches|in|''|\"|cm|and", "") %>%  #remove inches and other symbols
    str_replace("^([4-7])\\s*[,\\.\\s+]\\s*(\\d*)$", "\\1'\\2") %>% #change x.y, x,y x y
```

```r
    str_replace("^([56])'?$", "\\1'0") %>% #add 0 when to 5 or 6
    str_replace("^([12])\\s*,\\s*(\\d*)$", "\\1\\.\\2") %>% #change european decimal
    str_trim() #remove extra space
}
converted <- problems %>% words_to_numbers %>% convert_format
remaining_problems <- converted[not_inches_or_cm(converted)]
pattern <- "^[4-7]\\s*'\\s*\\d+\\.?\\d*$"
index <- str_detect(remaining_problems, pattern)
remaining_problems[!index]
```

```
##  [1] "511"       "2"         ">9000"     "11111"     "103.2"
##  [6] "19"        "300"       "7"         "214"       "0.7"
## [11] "2'33"      "612"       "1.70"      "87"        "111"
## [16] "12"        "yyy"       "89"        "34"        "25"
## [21] "22"        "684"       "1"         "1"         "6*12"
## [26] "87"        "1.6"       "120"       "120"       "23"
## [31] "1.7"       "86"        "708,661"   "649,606"   "10000"
## [36] "1"         "728,346"   "0"         "100"       "88"
## [41] "7,283,465" "34"
```

Separating with Regex

It is also possible to extract data using regex.

As you've seen, you can separate the feet and inches into two columns using `separate`.

```r
s <- c("5'10", "6'1")
tab <- data.frame(x = s)
tab %>% separate(x, c("feet", "inches"), sep = "'")
```

```
##   feet inches
## 1    5     10
## 2    6      1
```

This can also be accomplished with regex using the `extract` function from the `tidyr` package.

```r
tab %>% extract(x, c("feet", "inches"), regex = "(\\d)'(\\d{1,2})")
```

```
##   feet inches
## 1    5     10
## 2    6      1
```

This is useful because regex provides a lot more flexibility, such as in the following situation, where separate fails.

```r
s <- c("5'10", "6'1\"", "5'8inches")
tab <- data.frame(x = s)
tab %>% separate(x, c("feet", "inches"), sep = "'")
```

```
##   feet  inches
## 1    5      10
## 2    6      1"
## 3    5 8inches
```

With a more complex regex, we can still extract the numbers.

```r
tab %>% extract(x, c("feet", "inches"), regex = "(\\d)'(\\d{1,2})")
```

```
##   feet inches
## 1    5     10
```

```
## 2    6      1
## 3    5      8
```

Putting it all Together

Here's the final product.

```
pattern <- "^([4-7])\\s*'\\s*(\\d+\\.?\\d*)$" #the regex pattern

smallest <- 50
tallest <- 84
new_heights <- reported_heights %>%  #create new heights and clean up each of the issue types
  mutate(original = height,
         height = words_to_numbers(height) %>% convert_format()) %>%
  extract(height, c("feet", "inches"), regex = pattern, remove = FALSE) %>%
  mutate_at(c("height", "feet", "inches"), as.numeric) %>%
  mutate(guess = 12*feet + inches) %>%
  mutate(height = case_when(
    !is.na(height) & between(height, smallest, tallest) ~ height, #inches
    !is.na(height) & between(height/2.54, smallest, tallest) ~ height/2.54, #centimeters
    !is.na(height) & between(height*100/2.54, smallest, tallest) ~ height*100/2.54, #meters
    !is.na(guess) & inches < 12 & between(guess, smallest, tallest) ~ guess, #feet'inches
    TRUE ~ as.numeric(NA))) %>%
  select(-guess)
```

```
## Warning in eval(substitute(expr), envir, enclos): NAs introduced by
## coercion
```

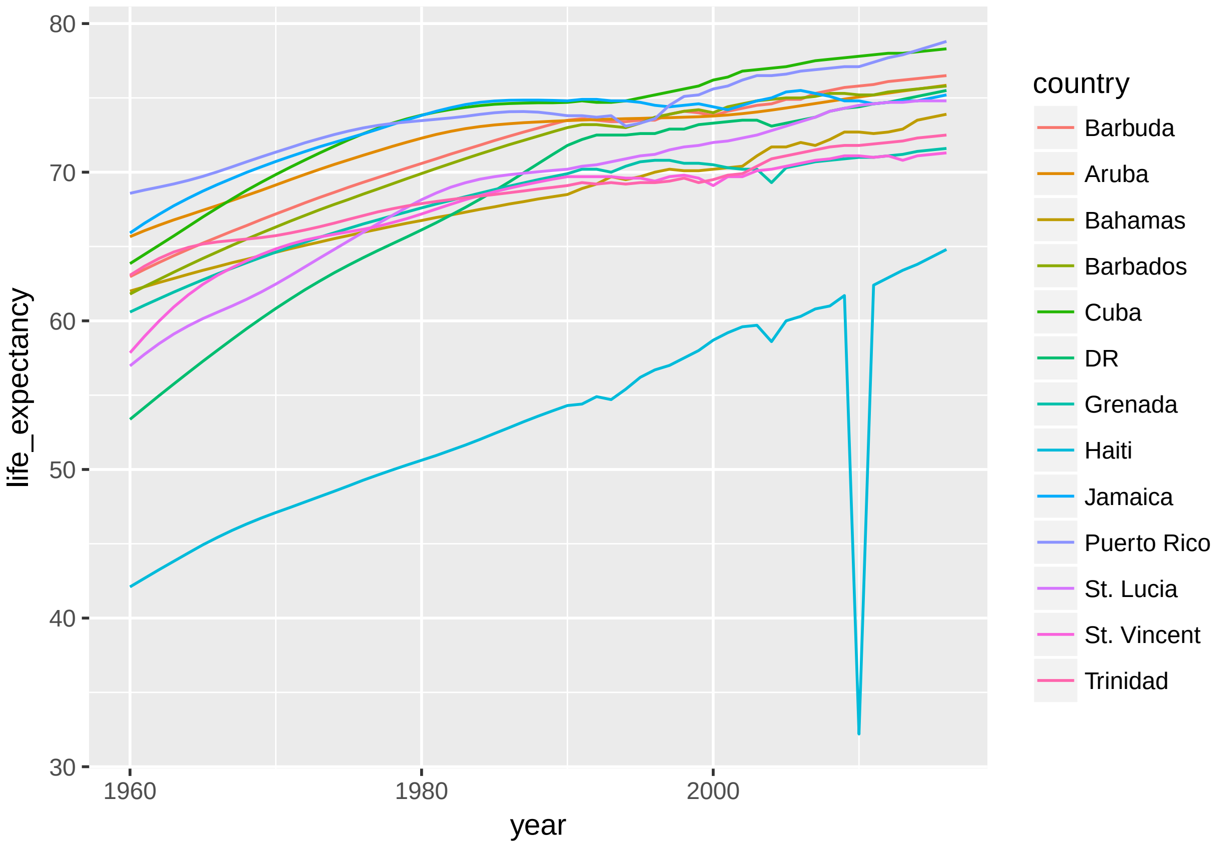
```
new_heights %>% # check all entries converted
  filter(not_inches(original)) %>%
  select(original, height) %>%
  arrange(height) %>%
  View()
```

Recoding

You can recode character names using `case_when`, but the `tidyverse` offers a function specifically designed for this task, `recode`.

Here's how you would implement it in (for example) the gapminder dataset.

```
data("gapminder")
gapminder %>% filter(region == "Caribbean") %>%
  mutate(country = recode(country,
                          'Antigua and Barbuda' = "Barbuda",
                          'Dominican Republic' = "DR",
                          'St. Vincent and the Grenadines' = "St. Vincent",
                          'Trinidad and Tobago' = "Trinidad")) %>%
  ggplot(aes(year, life_expectancy, color = country)) + geom_line()
```



Dates, Times, and Text Mining

Dates

The `tidyverse` includes functionality for dates in the `lubridate` package.

```
set.seed(2)
dates <- sample(polls_us_election_2016$startdate, 10) %>% sort()
dates
```

```
##  [1] "2015-11-09" "2016-02-15" "2016-07-09" "2016-08-12" "2016-08-17"
##  [6] "2016-09-18" "2016-10-04" "2016-10-10" "2016-10-18" "2016-10-25"
```

The functions `year`, `month`, and `day` extract those values. `month` can also extract month labels when label is set to TRUE.

```
library(lubridate)
```

```
##
## Attaching package: 'lubridate'

## The following object is masked from 'package:base':
##
##     date
```

```
data.frame(date = days(dates),
           month = month(dates),
           day = day(dates),
           year = year(dates)) %>% head()
```

```
##                date month day year
## 1 16748d 0H 0M 0S    11   9 2015
## 2 16846d 0H 0M 0S     2  15 2016
## 3 16991d 0H 0M 0S     7   9 2016
## 4 17025d 0H 0M 0S     8  12 2016
## 5 17030d 0H 0M 0S     8  17 2016
## 6 17062d 0H 0M 0S     9  18 2016
```

`ymd` will convert most kinds of strings that include year, month, and day in that order. All of the other orders are functions as well: `ydm`, `myd`, `dmy`, and `dym`.

The preferred format for dates is the *ISO 8601* format, which is **YYYY-MM-DD**.

Times

Lubridate also provides a function that gives the time zone: `now`. You can extract the hours, minutes, and seconds with `hour`, `minute`, and `second`.

`hms` can parse strings into times, and `mdy_hms` (and its variants) can parse dates and times that are in the same string.

Text Mining