MFE R Programming Workshop

Week 6

Brett Dunn and Mahyar Kargar

Fall 2017

Introduction

Questions

Any questions before we start?

Overview

- `%>%`
- `tidyr`
- `dplyr`

%>%

The Pipe Operator %>%

- The `magnittr` package provides a pipe operator.
- See `vignette("magrittr")`.
- Basic piping:
  - `x %>% f` is equivalent to `f(x)`
  - `x %>% f(y)` is equivalent to `f(x, y)`
  - `x %>% f %>% g %>% h` is equivalent to `h(g(f(x)))`
- The argument placeholder:
  - `x %>% f(y, .)` is equivalent to `f(y, x)`
  - `x %>% f(y, z = .)` is equivalent to `f(y, z = x)`

Expose the variables with %$%

- The %$% allows variable names (e.g. column names) to be used in a function.

```
library(magrittr)
iris %>%
  subset(Sepal.Length > mean(Sepal.Length)) %$%
  cor(Sepal.Length, Sepal.Width)
```

```
## [1] 0.3361992
```

Compound assignment pipe operations with %<>%

- There is also a pipe operator which can be used as shorthand notation in situations where the left-hand side is being "overwritten":

```
iris$Sepal.Length <-
  iris$Sepal.Length %>%
  sqrt()
```

Use the %<>% operator to avoid the repetition:

```
iris$Sepal.Length %<>% sqrt
```

- This operator works exactly like %>%, except the pipeline assigns the result rather than returning it.

`tidyr`

Hadley Wickham

- Hadley Wickham is practically famous in the R world
- He's developed a very large number of useful packages, e.g. `ggplot2` and `lubridate`.
- Today we will look at `dplyr` and `tidyr`.
- Tidy data is data that's easy to work with: it's easy to munge (with `dplyr`), visualise (with `ggplot2` or `ggvis`) and model (with `R`'s hundreds of modelling packages).
- The two most important properties of tidy data are:
  - Each column is a variable.
  - Each row is an observation.
- Check R for Data Science book.

Sample data

- A common problem is a dataset where some of the column names are not names of variables, but values of a variable.
- Take `table4a`: the column names 1999 and 2000 represent values of the `year` variable, and each row represents two observations, not one.
- `tidyr` is a member of the core `tidyverse`.

```
library(tidyverse)
table4a
```

```
## # A tibble: 3 × 3
##       country `1999` `2000`
## *       <chr>  <int>  <int>
## 1 Afghanistan    745   2666
## 2      Brazil  37737  80488
## 3       China 212258 213766
```

Bring columns together with `gather()`

- To tidy a dataset like this, we need to **gather** those columns into a new pair of variables. To describe that operation we need three parameters:
  - The set of columns that represent values, not variables. In this example, those are the columns 1999 and 2000.
  - The name of the variable whose values form the column names. I call that the `key`, and here it is `year`.
  - The name of the variable whose values are spread over the cells. I call that `value`, and here it's the number of `cases`.

Bring columns together with gather()

- In the final result, the gathered columns are dropped, and we get new `key` and `value` columns.

```r
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
```

```
## # A tibble: 6 × 3
##       country  year  cases
##         <chr> <chr>  <int>
## 1 Afghanistan  1999    745
## 2      Brazil  1999  37737
## 3       China  1999 212258
## 4 Afghanistan  2000   2666
## 5      Brazil  2000  80488
## 6       China  2000 213766
```

Split a column with spread()

- Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows.
- For example, take table2: an observation is a country in a year, but each observation is spread across two rows.

```
table2
```

```
## # A tibble: 12 × 4
##        country  year       type      count
##          <chr> <int>      <chr>      <int>
## 1  Afghanistan  1999      cases        745
## 2  Afghanistan  1999 population   19987071
## 3  Afghanistan  2000      cases       2666
## 4  Afghanistan  2000 population   20595360
## 5       Brazil  1999      cases      37737
## 6       Brazil  1999 population  172006362
## 7       Brazil  2000      cases      80488
## 8       Brazil  2000 population  174504898
```

`spreading`

- To tidy this up, we first analyse the representation in similar way to `gather()`. This time, however, we only need two parameters:
  - The column that contains variable names, the `key` column. Here, it's `type`.
  - The column that contains values forms multiple variables, the `value` column. Here it's `count`.

spreading

- Once we've figured that out, we can use `spread()`, as shown below

```r
spread(table2, key = type, value = count)
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *       <chr> <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3      Brazil  1999  37737  172006362
## 4      Brazil  2000  80488  174504898
## 5       China  1999 212258 1272915272
## 6       China  2000 213766 1280428583
```

spread() and gather() are complements

```r
df <- data.frame(x = c("a", "b"), y = c(3, 4),
                 z = c(5, 6))
df
```

```
##   x y z
## 1 a 3 5
## 2 b 4 6
```

```r
df %>% spread(x, y) %>% gather(x, y, a:b, na.rm = TRUE)
```

```
##   z x y
## 1 5 a 3
## 4 6 b 4
```

There's much more

- As usual, read the vignette on the CRAN page
- Also check Chapter 12 of R for Data Science book.

dplyr

Overview of `dplyr`

- `dplyr` provides a grammar of data manipulation.
  - A simple way to interact with data.
- We learn about:
  - tibble structure `tbl`
  - The pipe operator `%>%`
  - Using `dplyr` with databases
- The [`dplyr` introduction vignette] is a good resource.

`dplyr` and `data.table`

- See this post.
- Here are my thoughts:
  - For data less than 1 million rows, it is reported that there is not a significant speed difference between the two.
  - For large data that can fit in memory, use `data.table`.
  - For data than cannot fit in memory, you could use `dplyr` with a database backend.
- `dtplyr` is a package to use `dplyr` with `data.table`.
  - It is slower than just using `data.table`.

Data: nycflights13

- To explore the basic data manipulation verbs of dplyr, we'll start with the built in ‘nycflights13} data frame
- This dataset contains all flights that departed from New York City in 2013

```
library(dplyr)
library(nycflights13)
```

```
head(flights,4)
```

```
## # A tibble: 4 × 19
##    year month   day dep_time sched_dep_time dep_delay
##   <int> <int> <int>    <int>          <int>     <dbl>
## 1  2013     1     1      517            515         2
## 2  2013     1     1      533            529         4
## 3  2013     1     1      542            540         2
## 4  2013     1     1      544            545        -1
## # ... with 13 more variables: arr_time <int>,
```

tbls (Tibbles)

- A tbl will only display the data that will fit in your console. -glimpse() is another nice way to look at the data

```
flights <- tbl_df(flights)
flights
```

```
## # A tibble: 336,776 × 19
##     year month   day dep_time sched_dep_time dep_delay
##    <int> <int> <int>    <int>          <int>     <dbl>
## 1   2013     1     1      517            515         2
## 2   2013     1     1      533            529         4
## 3   2013     1     1      542            540         2
## 4   2013     1     1      544            545        -1
## 5   2013     1     1      554            600        -6
## 6   2013     1     1      554            558        -4
## 7   2013     1     1      555            600        -5
## 8   2013     1     1      557            600        -3
## 9   2013     1     1      557            600        -3
```

Single Table Verbs

- `dplyr` aims to provide a function for each basic verb of data manipulation:
- `select()` (and `rename()`)
  - returns a subset of the columns
- `filter()` (and `slice()`)
  - returns a subset of the rows
- `arrange()` - reorders rows
  - reorders the rows according to single or multiple variables
- `distinct()`
- `mutate()` (and `transmute()`)
  - builds adds new columns from the data
- `summarise()` - calculates summary statistics
  - which reduces each group to a single row by calculating aggregate measures
- `sample_n()` and `sample_frac()`

Tidy Data

- `dplyr` works best when variables are in columns and observations are in rows.
- You can use `tidyr` to help you create a tidy dataset.

Select Columns by Name with `select()`

- `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions:

```
# Select columns by name
select(flights, year, month, day)
```

```
## # A tibble: 336,776 × 3
##      year month   day
##     <int> <int> <int>
## 1   2013     1     1
## 2   2013     1     1
## 3   2013     1     1
## 4   2013     1     1
## 5   2013     1     1
## 6   2013     1     1
## 7   2013     1     1
## 8   2013     1     1
```

Select a Range of Columns with :

```r
# Select all columns between year and day (inclusive)
select(flights, year:day)
```

```
## # A tibble: 336,776 × 3
##     year month   day
##    <int> <int> <int>
## 1   2013     1     1
## 2   2013     1     1
## 3   2013     1     1
## 4   2013     1     1
## 5   2013     1     1
## 6   2013     1     1
## 7   2013     1     1
## 8   2013     1     1
## 9   2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

An Example of -(col1:col2)

```
# Select all columns except those from year to day (inclus
select(flights, -(year:day))
```

```
## # A tibble: 336,776 × 16
##    dep_time sched_dep_time dep_delay arr_time
##       <int>          <int>     <dbl>    <int>
## 1       517            515         2      830
## 2       533            529         4      850
## 3       542            540         2      923
## 4       544            545        -1     1004
## 5       554            600        -6      812
## 6       554            558        -4      740
## 7       555            600        -5      913
## 8       557            600        -3      709
## 9       557            600        -3      838
## 10      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>
```

`select` Helper Functions

- dplyr comes with a set of helper functions that can help you select groups of variables inside a `select()` call:
- `starts_with("X")`: every name that starts with "X",
- `ends_with("X")`: every name that ends with "X",
- `contains("X")`: every name that contains "X",
- `matches("X")`: every name that matches "X", where "X" can be a regular expression,
- `num_range("x", 1:5)`: the variables named x01, x02, x03, x04 and x05,
- `one_of(x)`: every name that appears in x, which should be a character vector.

Add New Columns with mutate()

```r
mutate(flights,
       gain = arr_delay - dep_delay,
       speed = distance / air_time * 60)
```

```
## # A tibble: 336,776 × 21
##     year month   day dep_time sched_dep_time dep_delay
##    <int> <int> <int>    <int>          <int>     <dbl>
## 1   2013     1     1      517            515         2
## 2   2013     1     1      533            529         4
## 3   2013     1     1      542            540         2
## 4   2013     1     1      544            545        -1
## 5   2013     1     1      554            600        -6
## 6   2013     1     1      554            558        -4
## 7   2013     1     1      555            600        -5
## 8   2013     1     1      557            600        -3
## 9   2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## # ... with 336,766 more rows, and 15 more variables:
```

If you only want to keep the new variables, use `transmute()`

```
transmute(flights,
          gain = arr_delay - dep_delay,
          gain_per_hour = gain / (air_time / 60)
)
```

```
## # A tibble: 336,776 × 2
##     gain gain_per_hour
##    <dbl>         <dbl>
## 1      9      2.378855
## 2     16      4.229075
## 3     31     11.625000
## 4    -17     -5.573770
## 5    -19     -9.827586
## 6     16      6.400000
## 7     24      9.113924
## 8    -11    -12.452830
## 9      5      2.142857
```

Filter rows with `filter()`

- `filter()` allows you to select a subset of rows in a data frame.
- The first argument is the name of the data frame.
- The second and subsequent arguments are the expressions that filter the data frame
- Select all flights on January 1st with:

```
filter(flights, month == 1, day == 1)
```

```
## # A tibble: 842 × 19
##     year month   day dep_time sched_dep_time dep_delay
##    <int> <int> <int>    <int>          <int>     <dbl>
## 1   2013     1     1      517            515         2
## 2   2013     1     1      533            529         4
## 3   2013     1     1      542            540         2
## 4   2013     1     1      544            545        -1
## 5   2013     1     1      554            600        -6
## 6   2013     1     1      554            558        -4
## 7   2013     1     1      555            600        -5
```

Select rows by position

- To select rows by position, use `slice()`

```
slice(flights, 1:10)
```

```
## # A tibble: 10 × 19
##     year month   day dep_time sched_dep_time dep_delay
##    <int> <int> <int>    <int>          <int>     <dbl>
## 1   2013     1     1      517            515         2
## 2   2013     1     1      533            529         4
## 3   2013     1     1      542            540         2
## 4   2013     1     1      544            545        -1
## 5   2013     1     1      554            600        -6
## 6   2013     1     1      554            558        -4
## 7   2013     1     1      555            600        -5
## 8   2013     1     1      557            600        -3
## 9   2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## # ... with 13 more variables: arr_time <int>,
```

Arrange rows with arrange()

- `arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them.

```
arrange(flights, year, month, day)
```

```
## # A tibble: 336,776 × 19
##      year month   day dep_time sched_dep_time dep_delay
##     <int> <int> <int>    <int>          <int>     <dbl>
## 1    2013     1     1      517            515         2
## 2    2013     1     1      533            529         4
## 3    2013     1     1      542            540         2
## 4    2013     1     1      544            545        -1
## 5    2013     1     1      554            600        -6
## 6    2013     1     1      554            558        -4
## 7    2013     1     1      555            600        -5
## 8    2013     1     1      557            600        -3
## 9    2013     1     1      557            600        -3
## 10   2013     1     1      558            600        -2
```

Use desc() to order a column in descending order

```
arrange(flights, desc(arr_delay))
```

```
## # A tibble: 336,776 × 19
##     year month   day dep_time sched_dep_time dep_delay
##    <int> <int> <int>    <int>          <int>     <dbl>
## 1   2013     1     9      641            900      1301
## 2   2013     6    15     1432           1935      1137
## 3   2013     1    10     1121           1635      1126
## 4   2013     9    20     1139           1845      1014
## 5   2013     7    22      845           1600      1005
## 6   2013     4    10     1100           1900       960
## 7   2013     3    17     2321            810       911
## 8   2013     7    22     2257            759       898
## 9   2013    12     5      756           1700       896
## 10  2013     5     3     1133           2055       878
## # ... with 336,766 more rows, and 13 more variables:
## #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl
## #   carrier <chr>, flight <int>, tailnum <chr>,
```

You can rename variables with `rename()`

```r
rename(flights, tail_num = tailnum)
```

```
## # A tibble: 336,776 × 19
##     year month   day dep_time sched_dep_time dep_delay
##    <int> <int> <int>    <int>          <int>     <dbl>
## 1   2013     1     1      517            515         2
## 2   2013     1     1      533            529         4
## 3   2013     1     1      542            540         2
## 4   2013     1     1      544            545        -1
## 5   2013     1     1      554            600        -6
## 6   2013     1     1      554            558        -4
## 7   2013     1     1      555            600        -5
## 8   2013     1     1      557            600        -3
## 9   2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## # ... with 336,766 more rows, and 13 more variables:
## #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl
## #   carrier <chr>, flight <int>, tail_num <chr>,
```

Extract distinct (unique) rows

- A common use of `select()` is to find the values of a set of variables.
- This is particularly useful in conjunction with the `distinct()` verb

```
distinct(select(flights, tailnum))
```

```
## # A tibble: 4,044 × 1
##    tailnum
##      <chr>
## 1   N14228
## 2   N24211
## 3   N619AA
## 4   N804JB
## 5   N668DN
## 6   N39463
## 7   N516JB
## 8   N829AS
```

Summarise values with summarise()

- The last verb is summarise(). It collapses a data frame to a single row.
- You can use any function you like in summarise() so long as the function can take a vector of data and return a single number.

```
summarise(flights,
          delay = mean(dep_delay, na.rm = TRUE))
```

```
## # A tibble: 1 × 1
##       delay
##       <dbl>
## 1 12.63907
```

dplyr aggregate functions

- dplyr provides several helpful aggregate functions of its own, in addition to the ones that are already defined in R. These include:
  - first(x) - The first element of vector x.
  - last(x) - The last element of vector x.
  - nth(x, n) - The nth element of vector x.
  - n() - The number of rows in the data.frame or group of observations that summarise() describes.
  - n_distinct(x) - The number of unique values in vector x.

Chaining

- The dplyr API is functional — function calls don't have side-effects.
- You must always save their results. UGLY
- To get around this problem, dplyr provides the %>% operator
- `x %>% f(y)` turns into `f(x, y)`

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

```
## Adding missing grouping variables: `year`, `month`, `day

## Source: local data frame [49 x 5]
## Groups: year, month [11]
##
```

Commonalities

- The syntax and function of all these verbs are very similar:
- The first argument is a data frame.
- The subsequent arguments describe what to do with the data frame.
- The result is a new data frame
- Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

Grouped operations

- These verbs are useful on their own, but they become really powerful when you apply them to groups of observations
- In dplyr, you do this by with the `group_by()` function.
- It breaks down a dataset into specified groups of rows.

Grouped operations (cont.)

- Grouping affects the verbs as follows:
- grouped `select()` is the same as ungrouped `select()`, except that grouping variables are always retained.
- grouped `arrange()` orders first by the grouping variables
- `mutate()` and `filter()` are most useful in conjunction with window functions (like `rank()`, or `min(x)` = x=). They are described in detail in `vignette("window-functions")`.
- `sample_n()` and `sample_frac()` sample the specified number/fraction of rows in each group.
- `slice()` extracts rows within each group.
- `summarise()` is powerful and easy to understand, as described in more detail below.

group_by Example

- For example, we could use these to find the number of planes and the number of flights that go to each possible destination:

```
flights %>%
  group_by(dest) %>%
  summarise(planes = n_distinct(tailnum),
            flights = n())
```

```
## # A tibble: 105 × 3
##     dest planes flights
##    <chr>  <int>   <int>
## 1    ABQ    108     254
## 2    ACK     58     265
## 3    ALB    172     439
## 4    ANC      6       8
## 5    ATL   1180   17215
## 6    AUS    993    2439
## 7    AVL    159     275
```

Multiple table verbs

- dplyr implements the four most useful SQL joins:
- `inner_join(x, y)`: matching x + y
- `left_join(x, y)`: all x + matching y
- `semi_join(x, y)`: all x with match in y
- `anti_join(x, y)`: all x without match in y
- And provides methods for:
- `intersect(x, y)`: all rows in both x and y
- `union(x, y)`: rows in either x or y
- `setdiff(x, y)`: rows in x, but not y

Joins from `dplyr` Map to SQL

- `inner_join(x, y)`
  - `SELECT * FROM x JOIN y ON x.a = y.a`
- `left_join(x, y)`
  - `SELECT * FROM x LEFT JOIN y ON x.a = y.a`
- `right_join(x, y)`
  - `SELECT * FROM x RIGHT JOIN y ON x.a = y.a`
- `full_join(x, y)`
  - `SELECT * FROM x FULL JOIN y ON x.a = y.a`
- `semi_join(x, y)`
  - `SELECT * FROM x WHERE EXISTS (SELECT 1 FROM y WHERE x.a = y.a)`
- `anti_join(x, y)`
  - `SELECT * FROM x WHERE NOT EXISTS (SELECT 1 FROM y WHERE x.a = y.a)`

Lab 3

- Let's redo Lab 3 with `dplyr`.