Lesson 06 - Data Wrangling

Jul 18, 2016

In this lesson, we will cover new methods for data management and data analysis workflows.

For this lesson, the following will be helpful:

- Code
- Data set 1: Census data
- Data set 2: Airport data
- Data set 3: Country data

Basics of scientific workflows

Store original data in a text file

Raw data should be stored in a text file. By doing all post-processing in R, you avoid having

multiple versions of data files, which can cause confusion.

Create an R script to plot and process data

One of the biggest strengths of R over programs like Excel is reproducibility. Storing your analysis and plots in a script allows you to re-create your results easily–extremely helpful for situations such as publishing your data, writing a methods section, or receiving more data (such as another replicate) and incorporating it into your pipeline.

You can save plots via the RStudio Plot window, but you can also save plots to PDF (or png) within a script:

```
> pdf('iris_petal_lengths.pdf', width=7, height=7)
> boxplot(iris$Petal.Length ~ iris$Species)
> dev.off()
```

The `pdf()` command opens a new "device", in this case a document 7 x 7 inches, to which all new plots are created. This device is closed with the command `dev.off()`, after which plots will be created in the RStudio window.

Including these commands in your script can be useful for saving vector-based (infinite-resolution, editable in Illustrator or Inkscape for publication) graphics as part of your workflow.

README files

A text README file (named something such as `README` or `readme.txt`) in the parent directory of a project that explains what each file in that directory refers to is really helpful as a reminder for collaborators, or for your future self.

Version control

Version control allows you to keep track of changes to your project. You may be familiar with the basic version control used by Dropbox or Google Drive. Using a program such as **git** gives you more power–you can keep notes of why you made changes, try out branching analysis within a project, and more.

You can use git for a project in multiple ways (you may need to install it first):

- On the command line (`git init`)
- Within RStudio (File -> New Project -> New Directory -> Empty Project, making sure to check "Create a git repository")
- With a desktop client such as SourceTree

I will be talking about git more next week in the large group lecture, and there is also more information available in the Git book.

R packages

Base R has a lot of functionality, but sometimes you need different tools–one example would be to handle sequencing data. Instead of writing (and testing!)[1] these tools yourself, there is likely a package available with code that someone else has already written and tested. **Packages contain functions for accomplishing a set of related tasks.** (In this case, Rsamtools is a package that has functions for reading SAM/BAM files.)

- The CRAN repository is the main archive for R packages. CRAN contains trusted packages, and it's what R accesses when you call `install.packages()` without specifying a source.
- There are other repositories of R packages, such as Bioconductor, which is a great resource for packages to process biological data.
- You can also install individual packages from github with the devtools package, or by specifying a file source.

Packages come with a reference manual, which contains the help documentation for each command. They also come with vignettes about each package that you can access by their CRAN page or with the `browseVignettes` function (e.g. `browseVignettes(package = 'dplyr')`).

Today we will be using the tidyr and dplyr packages, two very common and useful R packages by Hadley Wickham.

Installing packages

You can install packages as so:

```
> install.packages('tidyr')
> install.packages('dplyr')
```

You may have to specify a CRAN mirror. It doesn't really matter which one you choose (in theory, a close one will be faster). If you get an error downloading the package, try using another mirror.

Some packages have **dependencies**, other packages that they need to run. For example, tidyr and dplyr both require the magrittr package to run. Installing dplyr (or tidyr) will install all of its dependencies, and loading one of packages will load all of their dependencies.

Loading packages

The functions in a package are not accessible until you load it into the environment. The `library` function loads packages:

```
> library(tidyr)
> library(dplyr)
```

You may get output such as:

```
Attaching package: ‘dplyr’

The following objects are masked from ‘package:stats’:

    filter, lag

The following objects are masked from ‘package:base’:

    intersect, setdiff, setequal, union

Warning message:
package ‘dplyr’ was built under R version 3.2.5
```

(or possibly different output). “The following objects are masked” means that there is a conflict in function names (the function `intersect` exists in both dplyr and base), and the dplyr function will take precedent.

The warning message “package ‘dplyr’ was built under R version 3.2.5” is probably irrelevant to your uses.

So, let’s start using these packages!

Tidy data with tidyr

Hadley’s tidy data vignette goes into excellent detail about what makes data “tidy”, or easy to work with.

In summary: A data frame is tidy when each column represents a variable, and each row represents an observation.

Structuring your data in a tidy way helps with coding and interpretation. You also need to have tidy data in order to use ggplot2 effectively, which we’ll talk about next lesson.

If you are working with other people’s data, you will probably spend a lot of time restructuring it to use it efficiently. The dplyr and tidyr packages have features that make restructuring and analyzing simpler.

Let’s look at some real-world untidy data: population estimates from the US Census.

```
> head(pop.estimates)
  SEX AGE CENSUS2010POP ESTIMATESBASE2010 POPESTIMATE2010
1   0   0       3944153           3944160         3951330
2   0   1       3978070           3978090         3957888
3   0   2       4096929           4096939         4090862
4   0   3       4119040           4119051         4111920
5   0   4       4063170           4063186         4077551
6   0   5       4056858           4056872         4064653
  POPESTIMATE2011 POPESTIMATE2012 POPESTIMATE2013
1         3963087         3926540         3931141
2         3966551         3977939         3942872
3         3971565         3980095         3992720
4         4102470         3983157         3992734
5         4122294         4112849         3994449
6         4087709         4132242         4123626
  POPESTIMATE2014 POPESTIMATE2015
1         3949775         3978038
2         3949776         3968564
3         3959664         3966583
4         4007079         3974061
5         4005716         4020035
6         4006900         4018158
```

So, each population estimate has its own column. This is **not tidy**. The data set has five variables: sex, age, actual population, estimated population, and year. However, each year has its own column, rather than having a column for the year and a column for the estimated population.

spread and gather

We can use the `gather` function in tidyr to transform this data into tidy data.

`gather` takes four arguments: data frame, key, value, and columns to transform.

```
> pop.estimates.tidy <- gather(pop.estimates, Year, EstPop,
+     c(POPESTIMATE2010, POPESTIMATE2011, POPESTIMATE2012,
+       POPESTIMATE2013, POPESTIMATE2014, POPESTIMATE2015))
> head(pop.estimates.tidy)
  SEX AGE CENSUS2010POP ESTIMATESBASE2010           Year  EstPop
1   0   0       3944153           3944160 POPESTIMATE2010 3951330
2   0   1       3978070           3978090 POPESTIMATE2010 3957888
3   0   2       4096929           4096939 POPESTIMATE2010 4090862
4   0   3       4119040           4119051 POPESTIMATE2010 4111920
5   0   4       4063170           4063186 POPESTIMATE2010 4077551
```

```
6   0   5       4056858           4056872 POPESTIMATE2010 4064653
```

The column names become the keys (in this case, “Year”), and the values are gathered into a single column (in this case, “EstPop”). Here, I specified the column names manually[2]. You may have noticed that **dplyr and tidyr use bare words for column names instead of strings**. This is a major difference compared to base R.

The `spread` function does exactly the opposite of the `gather` function: it spreads a key/value pair into new columns.

```
> pop.estimates.untidy <- spread(pop.estimates.tidy, Year, EstPop)
> head(pop.estimates.untidy)
  SEX AGE CENSUS2010POP ESTIMATESBASE2010 POPESTIMATE2010
1   0   0       3944153           3944160         3951330
2   0   1       3978070           3978090         3957888
3   0   2       4096929           4096939         4090862
4   0   3       4119040           4119051         4111920
5   0   4       4063170           4063186         4077551
6   0   5       4056858           4056872         4064653
  POPESTIMATE2011 POPESTIMATE2012 POPESTIMATE2013
1         3963087         3926540         3931141
2         3966551         3977939         3942872
3         3971565         3980095         3992720
4         4102470         3983157         3992734
5         4122294         4112849         3994449
6         4087709         4132242         4123626
  POPESTIMATE2014 POPESTIMATE2015
1         3949775         3978038
2         3949776         3968564
3         3959664         3966583
4         4007079         3974061
5         4005716         4020035
6         4006900         4018158
```

We turn each value in our Key column (Year) to a column, using the values in the Value column (EstPop), and we recreate our original data.

Piping

The magrittr package that both tidyr and dplyr load introduces a new pipe operator: `%>%`. This operator sends data from the last function into the next function.

For example, we can rewrite the `gather` function above as:

```
> pop.estimates %>%
+     gather(Year, EstPop,
+            c(POPESTIMATE2010, POPESTIMATE2011, POPESTIMATE2012,
+              POPESTIMATE2013, POPESTIMATE2014, POPESTIMATE2015))
```

The pipe function sends the previous data to become the first argument in the next function. We can use this feature to create chains of function calls:

```
> pop.estimates %>%
+     gather(Year, EstPop,
+            c(POPESTIMATE2010, POPESTIMATE2011, POPESTIMATE2012,
+              POPESTIMATE2013, POPESTIMATE2014, POPESTIMATE2015)) %>%
+     head
```

Functions don't have to belong to the tidyr or dplyr packages to use them with the pipe! You can pipe data into any function–in this case, the `head` function.

The tidyr and dplyr functions are all written so that chaining functions together is easy.

separate

In our `pop.estimates.tidy` data frame, we have a column for the year. However, this year is a character vector, and in a form we can't use for plotting. The `separate` function in tidyr lets us access this data.

```
> pop.estimates.tidy %>%
+     separate(Year, into=c('temp', 'Year2'), sep=-5, convert=TRUE) %>%
+     head
  SEX AGE CENSUS2010POP ESTIMATESBASE2010        temp Year2  EstPop
1   0   0       3944153           3944160 POPESTIMATE  2010 3951330
2   0   1       3978070           3978090 POPESTIMATE  2010 3957888
3   0   2       4096929           4096939 POPESTIMATE  2010 4090862
4   0   3       4119040           4119051 POPESTIMATE  2010 4111920
5   0   4       4063170           4063186 POPESTIMATE  2010 4077551
6   0   5       4056858           4056872 POPESTIMATE  2010 4064653
```

- The first argument in `separate` is the data frame (but here, it receives the data from the pipe).
- The second argument is the column to be separated–note again that it's a bare word, not in quotes.
- The third argument (`into`) is a character vector of the new column names. We don't need the column that says "POPESTIMATE", so I'm storing that in a temporary column for now.

- The fourth argument defines the separator. If this argument is not specified, `separate` splits at all alphanumeric values. Here, it's splitting at the 5th character from the end of the string. (For more details, see `?separate`.)
- `convert=TRUE` changes the data types of the new columns. In this case, the new "Year2" column is all numeric, so `tidyr` converts it to a numeric column. Very useful!
- Finally, the new data frame is piped to the `head` function.

If the data frame is being assigned to a variable, all the operations are carried out before assignment–in this case, the final data frame consists of only six rows.

Manipulating data with dplyr

The dplyr package makes basic data manipulations–filtering, sorting, renaming columns, removing columns–easier than in base R.

tbl_df / tibble

To start off with, dplyr introduces a new data structure called a "tibble" (formerly called a tbl_df). Tibbles are data frames, but with a few extra features, most notably smart printing.

We can turn a data frame into a tibble using the `as_data_frame` function (compare to base R's `as.data.frame`).

```
> as_data_frame(pop.estimates)
# A tibble: 306 x 10
     SEX   AGE CENSUS2010POP ESTIMATESBASE2010 POPESTIMATE2010
   <int> <int>         <int>              <int>           <int>
1      0     0       3944153            3944160         3951330
2      0     1       3978070            3978090         3957888
3      0     2       4096929            4096939         4090862
4      0     3       4119040            4119051         4111920
5      0     4       4063170            4063186         4077551
6      0     5       4056858            4056872         4064653
7      0     6       4066381            4066412         4073013
8      0     7       4030579            4030594         4043046
9      0     8       4046486            4046497         4025604
10     0     9       4148353            4148369         4125415
# ... with 296 more rows, and 5 more variables: POPESTIMATE2011 <int>,
#   POPESTIMATE2012 <int>, POPESTIMATE2013 <int>,
#   POPESTIMATE2014 <int>, POPESTIMATE2015 <int>
```

Tibbles only print out the first ten rows of a data frame (no more using `head` to peek at the data!

no more printing 5000 rows of a data frame on accident!), and they only print the columns that can fit on screen. These features make glancing at the data very easy.

Tibbles can be turned back into regular data frames using the `as.data.frame` function, but you probably won't find that necessary.

select

The `select` function allows us to select specific columns from the data frame. In `pop.estimates.tidy`, we have some columns that we aren't interested in–specifically "CENSUS2010POP" and "ESTIMATESBASE2010", and we can use `select` to get rid of them.

We can either select the columns we want:

```
> pop.estimates.refined <- pop.estimates.tidy %>%
+     as_data_frame %>%
+     separate(Year, into=c('temp', 'Year2'), sep=-5, convert=TRUE) %>%
+     select(SEX, AGE, Year2, EstPop)
> pop.estimates.refined
# A tibble: 1,836 x 4
     SEX   AGE Year2  EstPop
*  <int> <int> <int>   <int>
1      0     0  2010 3951330
2      0     1  2010 3957888
3      0     2  2010 4090862
4      0     3  2010 4111920
5      0     4  2010 4077551
6      0     5  2010 4064653
7      0     6  2010 4073013
8      0     7  2010 4043046
9      0     8  2010 4025604
10     0     9  2010 4125415
# ... with 1,826 more rows
```

Or remove the columns we don't want, by using a negative vector:

```
> pop.estimates.tidy %>%
+     as_data_frame %>%
+     separate(Year, into=c('temp', 'Year2'), sep=-5, convert=TRUE) %>%
+ select(-c(CENSUS2010POP, ESTIMATESBASE2010, temp))
```

The end results are the same. Again, note that we're using bare words here to describe these columns.

We can also use select to rename columns when selecting them:

```
> pop.estimates.refined %>%
+     select(Gender = SEX, Age = AGE, Year = Year2, EstPop)
```

Here, “EstPop” doesn’t change names, but all the other columns do.

mutate

The `mutate` function creates a new variable in a data frame. For example:

```
> population.change <- pop.estimates %>%
+     as_data_frame %>%
+     mutate(EstimatedChange = POPESTIMATE2015 - POPESTIMATE2010) %>%
+     select(SEX, AGE, EstimatedChange)
> population.change
# A tibble: 306 x 3
     SEX   AGE EstimatedChange
   <int> <int>           <int>
1      0     0           26708
2      0     1           10676
3      0     2         -124279
4      0     3         -137859
5      0     4          -57516
6      0     5          -46495
7      0     6          -53806
8      0     7          105314
9      0     8          142283
10     0     9            8149
# ... with 296 more rows
```

You can also use `mutate` to assign a single value to a column:

```
> population.change %>%
+     mutate(Country = 'United States')
# A tibble: 306 x 4
     SEX   AGE EstimatedChange       Country
   <int> <int>           <int>         <chr>
1      0     0           26708 United States
2      0     1           10676 United States
3      0     2         -124279 United States
4      0     3         -137859 United States
5      0     4          -57516 United States
```

```
6      0     5          -46495 United States
7      0     6          -53806 United States
8      0     7          105314 United States
9      0     8          142283 United States
10     0     9            8149 United States
# ... with 296 more rows
```

filter

`filter` removes rows based on logical criteria–the dplyr equivalent of `subset`. For example, let's say we're not interested in data by age or sex, and only want to select data for the entire population (coded as AGE == 999 and SEX == 0):

```
> pop.estimates.refined %>%
+     filter(AGE == 999 & SEX == 0)
# A tibble: 6 x 4
    SEX   AGE Year2    EstPop
  <int> <int> <int>     <int>
1     0   999  2010 309346863
2     0   999  2011 311718857
3     0   999  2012 314102623
4     0   999  2013 316427395
5     0   999  2014 318907401
6     0   999  2015 321418820
```

How could we look at the population for only people over 90?

group_by and summarize

The `group_by` function creates groups within a data frame, by one or by multiple variables.

Grouping makes it easy to perform within-group operations to create a new data frame, using the `summarize` (also `summarise`) or `mutate` functions.

```
> pop.estimates.refined %>%
+     filter(SEX != 0 & AGE != 999) %>%
+     group_by(SEX, Year2) %>%
+     summarize(Total = sum(EstPop))
Source: local data frame [12 x 3]
Groups: SEX [?]

     SEX Year2       Total
   <int> <int>       <int>
```

```
1      1  2010   152088043
2      1  2011   153291772
3      1  2012   154521077
4      1  2013   155706770
5      1  2014   156955337
6      1  2015   158229297
7      2  2010   157258820
8      2  2011   158427085
9      2  2012   159581546
10     2  2013   160720625
11     2  2014   161952064
12     2  2015   163189523
```

`summarize` collapses data into unique values of its groups, while `mutate` keeps all rows, but still performs operations over each group.

The `n()` function returns the number of rows. Since this census data has a number for each age, it's not very useful for our purposes, but we can use it for the airport data to find the number of airports per country:

```
> airports <- as_data_frame(read.csv('data/airports.csv',
>                                    stringsAsFactors=FALSE))
> airports %>%
+     group_by(country) %>%
+     summarize(NumAirports = n())
# A tibble: 240 x 2
               country NumAirports
                <fctr>       <int>
1          Afghanistan          21
2              Albania           1
3              Algeria          44
4       American Samoa           3
5               Angola          26
6             Anguilla           1
7           Antarctica          19
8  Antigua and Barbuda           2
9            Argentina         103
10             Armenia           4
# ... with 230 more rows
```

In this case, all combinations of gender and year have the same number of observations, but you may find this function useful in another data set.

Joining data frames

You may have the same columns in multiple data frames that you want to intersect. dplyr has functions to intersect these data frames together: `left_join`, `right_join`, `inner_join`, `full_join`, `semi_join`, `anti_join`. (The first three are the ones I use most frequently. For more information, click here.)

`left_join` is the most frequently-used join function. It takes a data frame, A, and joins all rows in B that match A. Unless you tell `left_join` which columns to join by (using the argument `by`), it will join by all columns that have the same names.

Taking the airport data for example: maybe you want to add a column for the currency of the host country, so that travelers know which currency they use. Using data from here, we can use `left_join` to add a row for currency:

```
> currency <- as_data_frame(read.csv('data/country-codes.csv',
+     stringsAsFactors=FALSE)) %>%
+     select(Country = name, Currency = ISO4217.currency_alphabetic_code)
> airport.currency <- airports %>%
+     select(airport.id, name, Country = country) %>%
+     left_join(currency)
Joining, by = "Country"
> airport.currency
# A tibble: 8,107 x 4
   airport.id                       name          Country Currency
        <int>                      <chr>            <chr>    <chr>
1           1                     Goroka Papua New Guinea      PGK
2           2                     Madang Papua New Guinea      PGK
3           3                 Mount Hagen Papua New Guinea      PGK
4           4                     Nadzab Papua New Guinea      PGK
5           5 Port Moresby Jacksons Intl Papua New Guinea      PGK
6           6                  Wewak Intl Papua New Guinea      PGK
7           7                  Narsarsuaq        Greenland      DKK
8           8                        Nuuk        Greenland      DKK
9           9          Sondre Stromfjord        Greenland      DKK
10         10              Thule Air Base        Greenland      DKK
# ... with 8,097 more rows
```

And more

There are many more functions in dplyr than what I covered here, that you might find useful for your data. The vignette is a good resource, as is the RStudio Data Wranging cheat sheet[3].

Homework

1. Create a new variable, Gender, based on the SEX column in `pop.estimates.refined`, except coded as a character or factor instead of as numeric. (Hint: `?ifelse` or `?factor` may be useful.) 0 refers to aggregated data (all), 1 refers to male, 2 refers to female.
2. Create two data frames with the following code. Join these two data frames with `inner_join`, `left_join`, `right_join`, and `full_join`. Based on the results, can you describe in plain English how each join function specifically works, and handles incomplete data?

```
df1 <- data.frame(
    Letter = c('A', 'C', 'E', 'G'),
    n1 = c(7, 8, 9, 10),
    stringsAsFactors=FALSE
)
df2 <- data.frame(
    Letter = c('A', 'B', 'C', 'D'),
    n2 = c(1, 2, 3, 4),
    stringsAsFactors=FALSE
)
```

Resources

- PLOS Computational Biology: Ten Simple Rules for Reproducible Computational Research
- PLOS Biology: Computing Workflows for Biologists: A Roadmap
- Tidy data
- Introduction to dplyr

1. Writing new tools is only half the battle–making sure the tools always behave as expected, even if they receive unexpected data, is a hugely important part of coding. ↩
2. dplyr's `starts_with` function might also come in handy for this data: `gather(pop.estimates, Year, EstPop, starts_with("POPESTIMATE"))`. ↩
3. I have this cheat sheet taped up next to my desk, and I look at it all the time when writing code. ↩

kutchko (UNC-Chapel Hill, Summer 2016)
TweetNTD