R Crash Course: Lesson 1

author: Aaron C Cochran date: November 07, 2017 autosize: false height: 1080 width: 1920

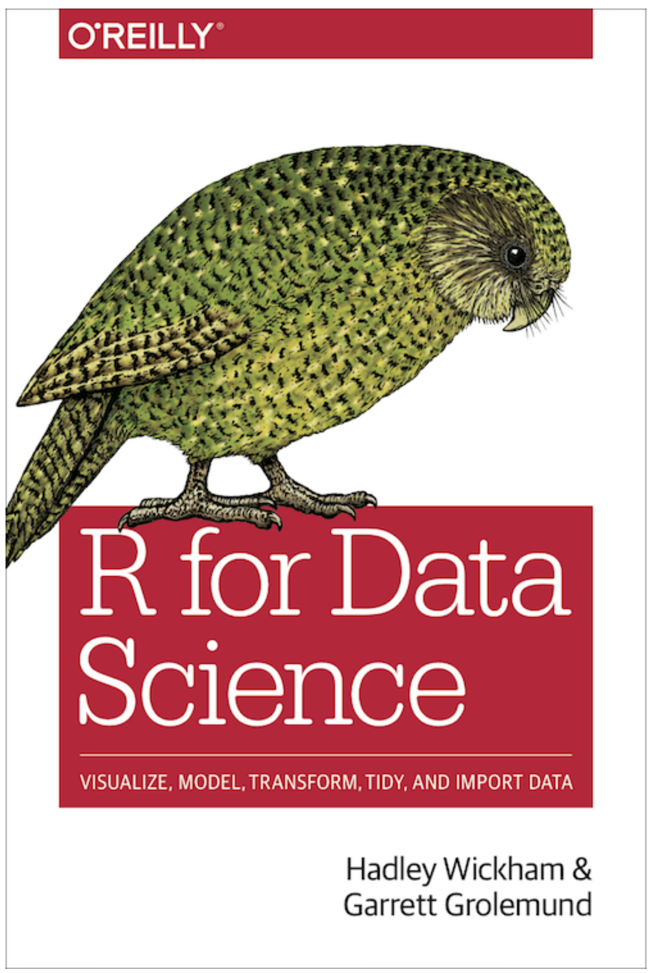
Introduction

We're going to pick this course up with the asssumption that you've already installed R and RStudio. If you need help doing that on your own work station, follow these links.

- The R Project for Statistical Computing https://cloud.r-project.org/
- RStudio Integrated Development Environment (IDE) https://www.rstudio.com/products/rstudio/download/

Note that: - R does not require admin access. You can install it directly to your Documents folder. - RStudio **DOES** require admin access to install. - RTools is optional, but allows you to use packages that need to be compiled before use.

== type:subsection We're going to primarily use the book **R for Data Science** by Hadley Wickham and Garret Grolemund.



http://r4ds.had.co.nx/index.html ***

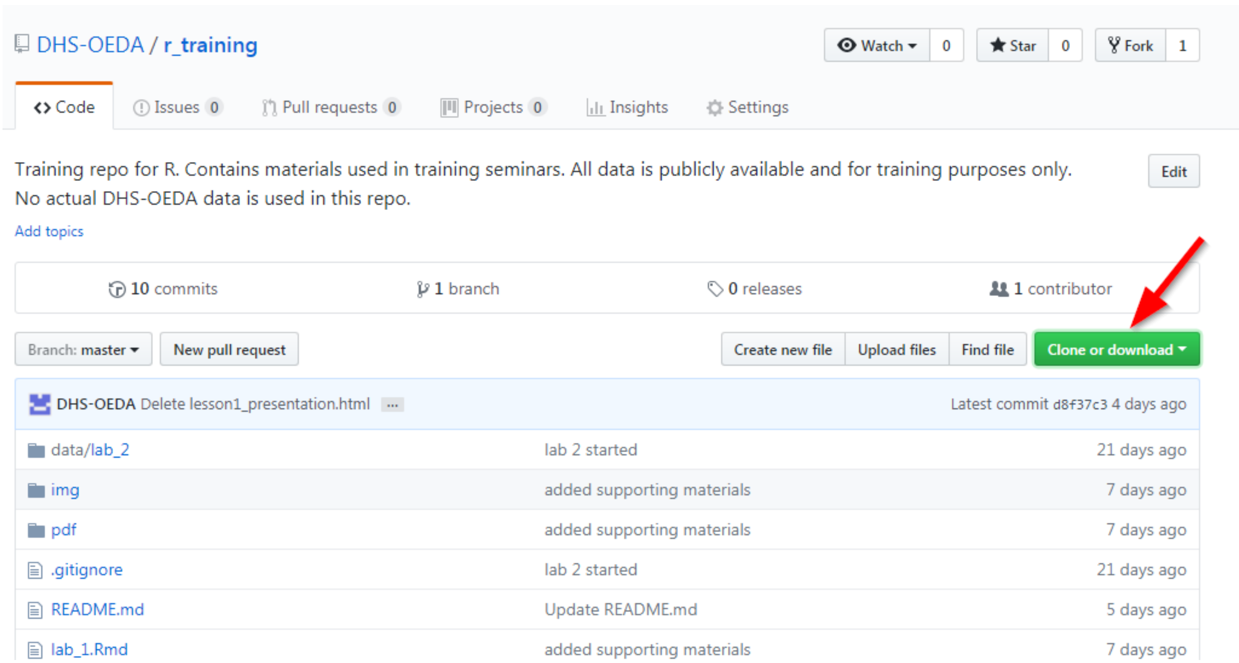
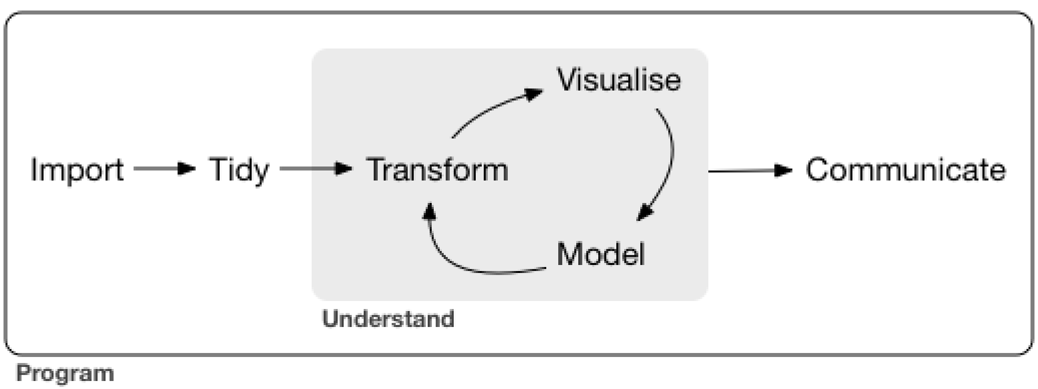


Figure 1: git-download

The very basics



***

- **Import** data into R
- **Tidy** the data for analysis
- **Transform** the data, **visualize** it, and use it to construct your **model**
- **Communicate** your results

Course Materials

Go here, download the repository contents as a zip:

https://www.github.com/DHS-OEDA/r_training

Then, extract it to a folder, and note the path to the folder.

Working Directory

Best practice: Create a new project folder for each project to isolate your code and data

File >> New Project...

Point it to the existing folder you just created.

```
getwd() # shows your current working directory
```

```
[1] "E:/R/r_training"
```

```
# setwd() # sets your working directory to a new location
```

Working Directory

Within your working directory, you can organize things how you like.

Every project I have in R has the following folders in the working directory:

- data *This contains any datasets in their unaltered form*

Beyond that, include what you like. Sometimes, I include documentation in a *docs* folder, or images in a *img* folder.

> If you consistently name your folders in your projects it keeps things sorted in your head. Data goes in data folder. Images go in img folder.

```
read_csv('data/mycsv.csv') # this is shorter and portable. It uses relative file paths.

read_csv('C:/Users/User/Documents/R/myproject/data/mycsv.csv') # this is bad. Not portable, and long.
```

Packages

R's base code is somewhat limited. The original language is from the mid-90's to early 2000's. You extend this functionality through packages. The main package here is `tidyverse` which is actually a collection of a number of packages and their dependencies that share a common philosophy.

```
install.packages("tidyverse")
```

You can also do it through the RStudio GUI under **Tools >> Install Packages**

Packages: The Tidyverse

Packages

Package help file

```
help(package="ggplot2")
```

Help with a function

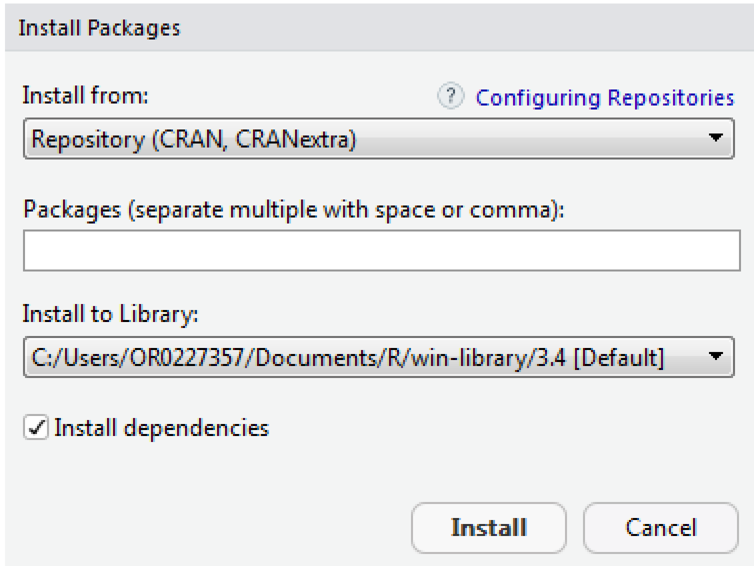


Figure 2: rstudio-packages

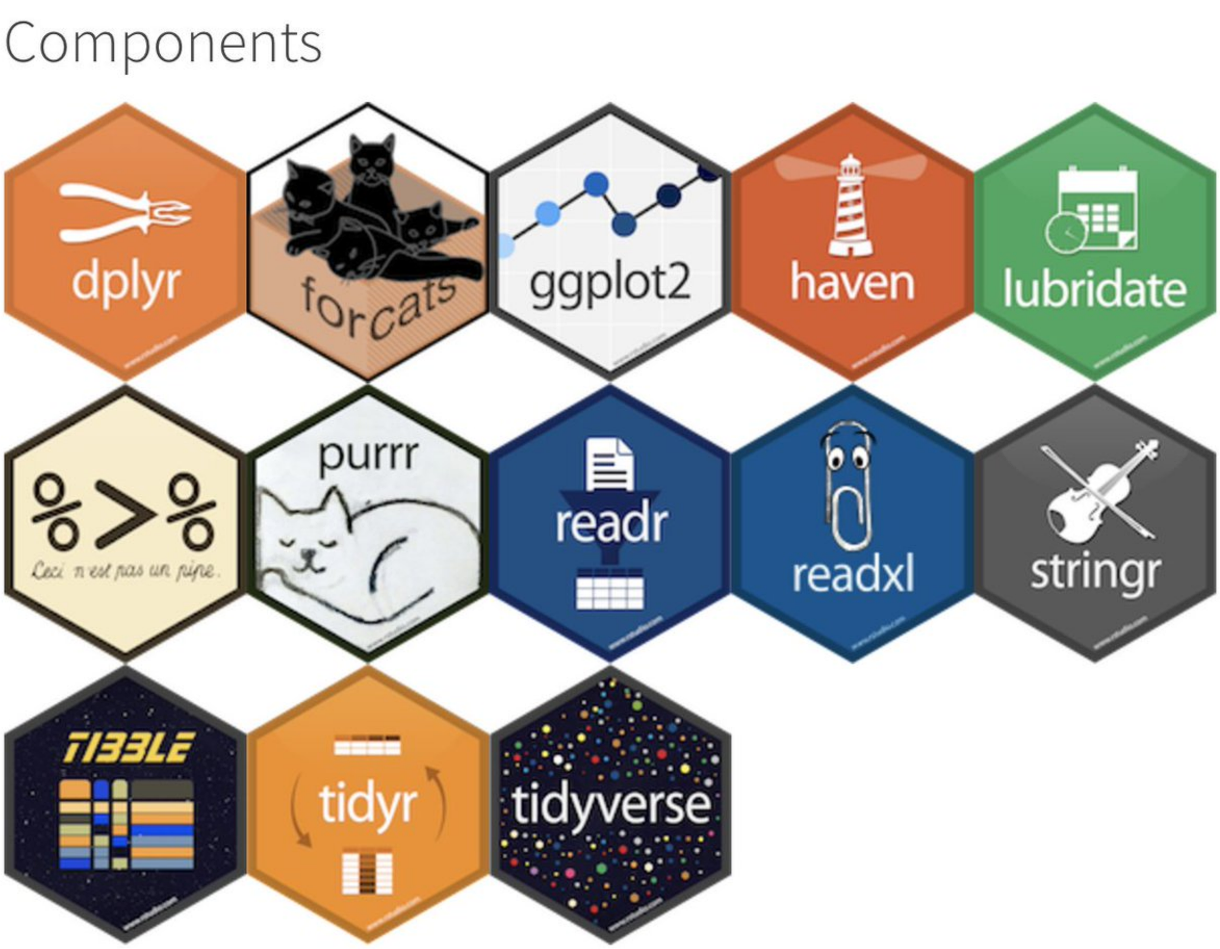


Figure 3: tidyverse

```r
# if you know the function and have the package loaded
?geom_point()
# if you can't find the function do this
??geom_point()
```

help files & vignettes

`table` function, for example...

```r
?table
```

Other packages

Beyond the so-called `tidyverse` we'll be using a dataset package called `nycflights13`.

```r
install.packages("nycflights13") # to install the package
library(nycflights13) # to load the package
```

Running R code

There are a few conventions we'll use in this course.

- Functions are in a code font and followed by parentheses, like `sum()` or `mean()`.
- Other R objects (like data or function arguments) are in a code font, without parentheses, like `flights` or `x`.
- If we want to be explicitly clear on what package we're using, we'll use the package name followed by two colons, like `dplyr::mutate()` or `nycflights13::flights`.

The last point is considered a best-practice when you are writing code to share, as it avoids ambiguity in naming. Often, packages might have functions that share the same name, and the last package you load will overwrite the other functions. Explicit naming avoids this overwriting.

Basics of R

type:section

R Console

The console is where you enter commands

```r
1+1
```

```
[1] 2
```

```r
print('Hello World')
```

```
[1] "Hello World"
```

```r
Sys.time()
```

```
[1] "2017-11-02 13:38:30 PDT"
```

Objects, Classes, and Methods

R is an object-oriented programming language. You give it commands to do things (*functions*) to objects.

R's math operations are *vectorized*. A *vector* is one class of object in R. Vectors are homogenous (only one type of "thing" in them).

```r
# a vector of numbers
c(1,2,3,4,5)
```

```
[1] 1 2 3 4 5
```

```r
# a vector of strings
c("This","is","a","vector","of","strings")
```

```
[1] "This"    "is"      "a"       "vector"  "of"      "strings"
```

```r
# this is a vector of booleans (and a missing value)
c(T,F,T,NA)
```

```
[1]  TRUE FALSE  TRUE    NA
```

Objects (continued)

A matrix is a n x m dimensional vector. It is also homogenous.

```r
matrix(seq(1:9), nrow=3, ncol=3)
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

===

You can select parts of a vector or matrix.

```r
m <- matrix(seq(1:9), nrow=3, ncol=3) # <- is the assignment operator

m[2, ] # select the 2nd row
```

```
[1] 2 5 8
```

```r
m[ , 1] # select the first column
```

```
[1] 1 2 3
```

```r
m[2,3] # select element in row 2, column 3
```

```
[1] 8
```

Data frames & Lists

Heterogenous data structures

```
a <- data.frame(
  "Col1"= seq(1:5),
  "Col2" = letters[c(1,2,3,4,5)],
  "Col3" = c(T,T,F,T,F)
)
a
```

```
  Col1 Col2  Col3
1    1    a  TRUE
2    2    b  TRUE
3    3    c FALSE
4    4    d  TRUE
5    5    e FALSE
```

==

Lists can even contain other data frames

```
l <- list(x=1:5, y=c('a', 'b'), z=a)
l['x'] # select the x element of the list (by name)
```

```
$x
[1] 1 2 3 4 5
```

```
l[[2]] # select the 2nd element of the list
```

```
[1] "a" "b"
```

```
l # see the whole list
```

```
$x
[1] 1 2 3 4 5

$y
[1] "a" "b"

$z
  Col1 Col2  Col3
1    1    a  TRUE
2    2    b  TRUE
3    3    c FALSE
4    4    d  TRUE
5    5    e FALSE
```

Tidyverse: tibbles

Tibbles are a modern take on data.frames. If you use `tidyverse` and the accompanying `tibble` package from the start you'll never notice the changes, but they make life easier.

- Displays data in a more friendly format
- They drop features that are no longer useful (`stringsAsFactors = FALSE`, I'm looking at you)
- Tibbles are faster to work with

```r
if (requireNamespace("microbenchmark")) {
  l <- replicate(26, sample(100), simplify = FALSE)
  names(l) <- letters

  microbenchmark::microbenchmark(
    as_tibble(l),
    as.data.frame(l)
  )
}
```

```
Unit: microseconds
             expr      min       lq     mean   median       uq       max
     as_tibble(l)  824.452 1075.033 1462.550 1181.394 1376.240 19499.014
 as.data.frame(l) 1005.327 1259.814 1662.608 1419.206 2033.039  4816.615
 neval
   100
   100
```

Reading & manipulating data

type:section

Using `readr` and `dplyr` from the `tidyverse` series of packages

==

Read in data from a .csv using `read_csv`.

```r
train <- read_csv('data/titanic/train.csv') # note the relative file path
head(train)
```

```
# A tibble: 6 x 12
  PassengerId Survived Pclass
        <int>    <int>  <int>
1           1        0      3
2           2        1      1
3           3        1      3
4           4        1      1
5           5        0      3
6           6        0      3
# ... with 9 more variables: Name <chr>, Sex <chr>, Age <dbl>,
#   SibSp <int>, Parch <int>, Ticket <chr>, Fare <dbl>, Cabin <chr>,
```

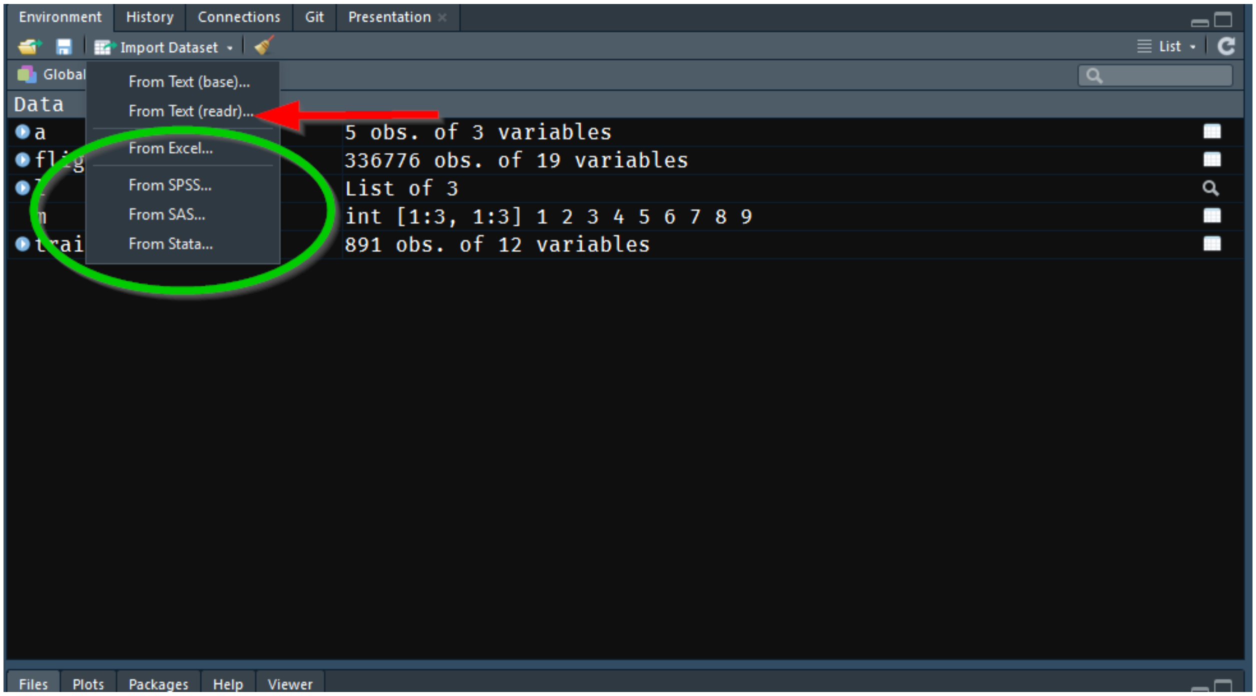


Figure 4: read-rstudio

```
#   Embarked <chr>
==================================================================================
```

You can also do this through RStudio's GUI

Data manipulation with `dplyr`

`dplyr` does a lot of things base R does, but faster, and with less code

`dplyr` uses "verbs":

- `arrange`: orders rows based on logic provided
- `mutate`: creates new variables from existing ones
- `filter`: returns a subset of the data.frame as a new table
- be sure to read `?base::Logic` and `?Comparison` for logical operators
- `select`: like filter, but for columns. Returns selected columns as a new table
- `summarise`: applies summary functions to columns and returns a new table
- `group_by`: creates a "grouped" version of the table and will do `dplyr` functions to each group individually

  Note: US English spelling or UK English spelling is fine

`tidyr`

In addition to `dplyr` there is `tidyr` which has more functions for manipulating data. It is also part of the `tidyverse` but worth mentioning in its own slide.

- `spread`: makes long (tidy) data wide
- `gather`: makes wide data long

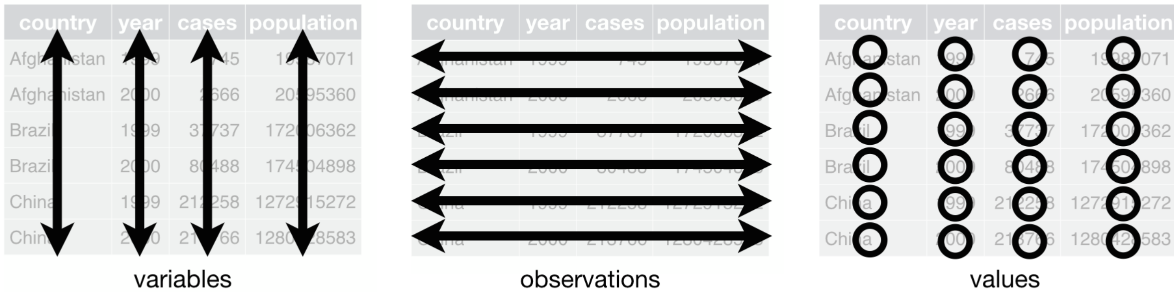


Figure 5: tidy-1

Principles of tidy data

> "Happy families are all alike; every unhappy family is unhappy in its own way" - Leo Tolstoy

Like Tolstoy's families, all tidy data is alike, but all untidy data is uniquely untidy.

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

   See https://www.jstatsoft.org/article/view/v059i10/v59i10.pdf by Hadley Wickham (2014) for more details

==

Our data

```
# let's create fake stock price data
set.seed(1) # pseudorandom numbers
stocks <- data.frame(
   time = as.Date('2009-01-01') + 0:9,
   X = rnorm(10, 20, 1),
   Y = rnorm(10, 20, 2),
   Z = rnorm(10, 20, 4)
 )
stocks
```

```
         time        X        Y        Z
1  2009-01-01 19.37355 23.02356 23.67591
2  2009-01-02 20.18364 20.77969 23.12855
3  2009-01-03 19.16437 18.75752 20.29826
4  2009-01-04 21.59528 15.57060 12.04259
5  2009-01-05 20.32951 22.24986 22.47930
6  2009-01-06 19.17953 19.91013 19.77549
7  2009-01-07 20.48743 19.96762 19.37682
8  2009-01-08 20.73832 21.88767 14.11699
9  2009-01-09 20.57578 21.64244 18.08740
10 2009-01-10 19.69461 21.18780 21.67177
```

We have 3 stocks: X, Y, Z. We have a time column with the times of the prices, and we have prices for each stock.

==

1. Each variable forms a column

Problem: Stock X, Y, and Z are variables as columns.

```
stocks_tidy <- # we're saving our data as a new data.frame
  gather(data=stocks, key= stock, value = price, X, Y, Z) # gathering the X, Y, and Z columns
head(stocks_tidy,12)
```

```
         time stock    price
1  2009-01-01     X 19.37355
2  2009-01-02     X 20.18364
3  2009-01-03     X 19.16437
4  2009-01-04     X 21.59528
5  2009-01-05     X 20.32951
6  2009-01-06     X 19.17953
7  2009-01-07     X 20.48743
8  2009-01-08     X 20.73832
9  2009-01-09     X 20.57578
10 2009-01-10     X 19.69461
11 2009-01-01     Y 23.02356
12 2009-01-02     Y 20.77969
```

===

Why does it matter if the data is tidy?

- vectorized math
- unified syntax between packages
- much faster for big data

`dplyr` pipeline: putting it together

`%>%` is the pipeline operator. It says "take the object on the left, and do the function on the right to it"

Synonymous to `f(x,y)` but makes for much easier to read code. For example, let's say we wanted to know the high and low value of each stock.

```
stocks %>% # raw data
  gather(stock, price, X:Z) %>% # made tidy
  group_by(stock) %>% # grouped by stock
  summarise(min = min(price), max=max(price)) # summary stats returned
```

```
# A tibble: 3 x 3
  stock      min      max
  <chr>    <dbl>    <dbl>
1     X 19.16437 21.59528
2     Y 15.57060 23.02356
3     Z 12.04259 23.67591
```

Another `dplyr` example

Taken from https://www.r-bloggers.com/introducing-tidyr/

```r
# make a dataset of messy data
set.seed(10)
messy <- data.frame(
  id = 1:4,
  trt = sample(rep(c('control', 'treatment'), each = 2)),
  work.T1 = runif(4), # ?runif if you want to know more
  home.T1 = runif(4), # but simply put it creates random deviates
  work.T2 = runif(4), # of a uniform distribution
  home.T2 = runif(4)
)
messy
```

```
  id       trt    work.T1   home.T1   work.T2    home.T2
1  1 treatment 0.08513597 0.6158293 0.1135090 0.05190332
2  2   control 0.22543662 0.4296715 0.5959253 0.26417767
3  3 treatment 0.27453052 0.6516557 0.3580500 0.39879073
4  4   control 0.27230507 0.5677378 0.4288094 0.83613414
```

==

First lets gather up our columns into a key-value pair (called `key` and `time` here)

```r
tidier <- messy %>% # take our original messy data
  gather(key, time, -id, -trt) # you can specify which columns to exclude by using the negative sign
tidier %>% head(8) # head() takes an argument to specify how many rows to show
```

```
  id       trt     key       time
1  1 treatment work.T1 0.08513597
2  2   control work.T1 0.22543662
3  3 treatment work.T1 0.27453052
4  4   control work.T1 0.27230507
5  1 treatment home.T1 0.61582931
6  2   control home.T1 0.42967153
7  3 treatment home.T1 0.65165567
8  4   control home.T1 0.56773775
```

===

Now let's split up that key into a location and time variable using regular expressions

> Regular expressions are a bear to learn, but very worth it in the end. One good tutorial is here https://regexone.com/

```r
tidy <- tidier %>%
  separate(key, into = c("location", "time"), sep = "\\.")
tidy %>% head(8)
```

```
  id       trt location time       time
1  1 treatment     work   T1 0.08513597
2  2   control     work   T1 0.22543662
3  3 treatment     work   T1 0.27453052
4  4   control     work   T1 0.27230507
5  1 treatment     home   T1 0.61582931
6  2   control     home   T1 0.42967153
7  3 treatment     home   T1 0.65165567
8  4   control     home   T1 0.56773775
```

dates and times in R

R stores dates and date-times.

`lubridate` from the `tidyverse` packages makes this much easier

```r
library(lubridate)
today()
```

```
[1] "2017-11-02"
```

```r
now()
```

```
[1] "2017-11-02 13:38:31 PDT"
```

string dates

Date/time often exists as strings in administrative databases.

```r
# using lubridate
ymd("2017-01-01")
```

```
[1] "2017-01-01"
```

```r
mdy("January 31st, 2017")
```

```
[1] "2017-01-31"
```

```r
dmy("31-Jan-2017")
```

```
[1] "2017-01-31"
```

dates and times

Using `nycflights13` flight data

```r
data(flights)

flights %>% select(year, month, day, hour, minute)
```

```
# A tibble: 336,776 x 5
    year month   day  hour minute
   <int> <int> <int> <dbl>  <dbl>
 1  2013     1     1     5     15
 2  2013     1     1     5     29
 3  2013     1     1     5     40
 4  2013     1     1     5     45
 5  2013     1     1     6      0
 6  2013     1     1     5     58
 7  2013     1     1     6      0
 8  2013     1     1     6      0
 9  2013     1     1     6      0
10  2013     1     1     6      0
# ... with 336,766 more rows
```

dates and times

```r
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))
```

```
# A tibble: 336,776 x 6
    year month   day  hour minute           departure
   <int> <int> <int> <dbl>  <dbl>              <dttm>
 1  2013     1     1     5     15 2013-01-01 05:15:00
 2  2013     1     1     5     29 2013-01-01 05:29:00
 3  2013     1     1     5     40 2013-01-01 05:40:00
 4  2013     1     1     5     45 2013-01-01 05:45:00
 5  2013     1     1     6      0 2013-01-01 06:00:00
 6  2013     1     1     5     58 2013-01-01 05:58:00
 7  2013     1     1     6      0 2013-01-01 06:00:00
 8  2013     1     1     6      0 2013-01-01 06:00:00
 9  2013     1     1     6      0 2013-01-01 06:00:00
10  2013     1     1     6      0 2013-01-01 06:00:00
# ... with 336,766 more rows
```

dates and times

Recall that the four time columns in `flights` were not date-time <`dttm`> variables.

```r
flights %>%
  select(year, month, day, hour, minute, dep_delay, arr_delay, dep_time, arr_time) %>%
  head()
```

```
# A tibble: 6 x 9
   year month   day  hour minute dep_delay arr_delay dep_time arr_time
  <int> <int> <int> <dbl>  <dbl>     <dbl>     <dbl>    <int>    <int>
1  2013     1     1     5     15         2        11      517      830
2  2013     1     1     5     29         4        20      533      850
3  2013     1     1     5     40         2        33      542      923
4  2013     1     1     5     45        -1       -18      544     1004
5  2013     1     1     6      0        -6       -25      554      812
6  2013     1     1     5     58        -4        12      554      740
```

dates and times

We need to use modulus arithmetic to pull out hours and minutes from the time variables. This is a bit more advanced, so let's walk through it slowly.

```r
# create a function -- for more information read the Functions chapter (19) in the text

make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100) # time %/% 100 extracts the hour, time %%
}
```

```
flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(year, month, day, sched_dep_time),
    sched_arr_time = make_datetime_100(year, month, day, sched_arr_time)) %>%
  select(origin, dest, ends_with("delay"), ends_with("time"))

flights_dt
```

```
# A tibble: 328,063 x 9
   origin  dest dep_delay arr_delay            dep_time
    <chr> <chr>     <dbl>     <dbl>              <dttm>
 1    EWR   IAH         2        11 2013-01-01 05:17:00
 2    LGA   IAH         4        20 2013-01-01 05:33:00
 3    JFK   MIA         2        33 2013-01-01 05:42:00
 4    JFK   BQN        -1       -18 2013-01-01 05:44:00
 5    LGA   ATL        -6       -25 2013-01-01 05:54:00
 6    EWR   ORD        -4        12 2013-01-01 05:54:00
 7    EWR   FLL        -5        19 2013-01-01 05:55:00
 8    LGA   IAD        -3       -14 2013-01-01 05:57:00
 9    JFK   MCO        -3        -8 2013-01-01 05:57:00
10    LGA   ORD        -2         8 2013-01-01 05:58:00
# ... with 328,053 more rows, and 4 more variables: sched_dep_time <dttm>,
#   arr_time <dttm>, sched_arr_time <dttm>, air_time <dbl>
```

why bother?

```
# do math with dttm

df <- flights_dt[1:10,]

df$travel_time <- df$arr_time - df$dep_time

df$travel_time
```

```
Time differences in hours
 [1] 3.216667 3.283333 3.683333 4.333333 2.300000 1.766667 3.300000
 [8] 1.200000 2.683333 1.916667
```

why bother?

ggplot knows how to work with dates and times

```
flights_dt %>%
  filter(dep_time < ymd(20130102)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 600, , size=1) # 600 s = 10 minutes
```

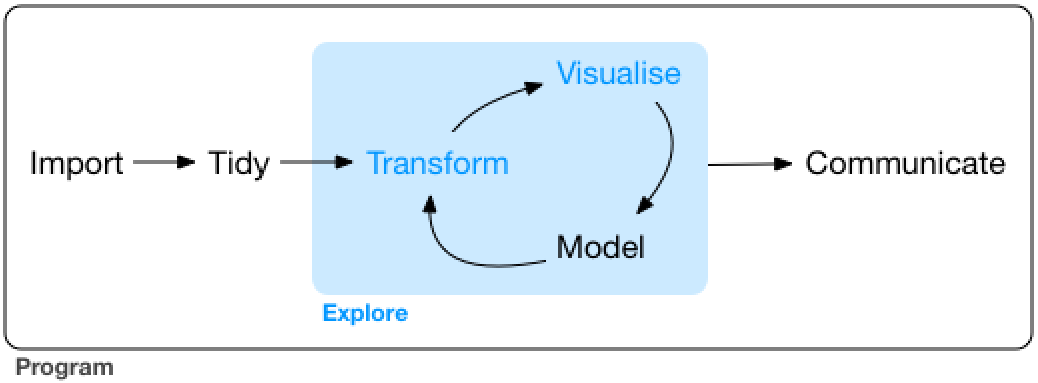


Figure 6: data science explore

Data exploration: Visualization

type:section

> "The simple graph has brought more information to the data analyst's mind than any other device." — John Tukey

Getting started

```
install.packages("tidyverse") # assuming you don't already have it installed
library(tidyverse) # load the package
```

Mpg data

We're going to use the `mpg` data frame found in ggplot2 (aka `ggplot2::mpg`). `mpg` contains observations collected by the EPA on 38 car models.

===

```
ggplot2::mpg
```

```
# A tibble: 234 x 11
   manufacturer      model displ  year   cyl      trans   drv   cty   hwy
          <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int>
 1         audi         a4   1.8  1999     4   auto(l5)     f    18    29
 2         audi         a4   1.8  1999     4 manual(m5)     f    21    29
 3         audi         a4   2.0  2008     4 manual(m6)     f    20    31
 4         audi         a4   2.0  2008     4   auto(av)     f    21    30
 5         audi         a4   2.8  1999     6   auto(l5)     f    16    26
 6         audi         a4   2.8  1999     6 manual(m5)     f    18    26
 7         audi         a4   3.1  2008     6   auto(av)     f    18    27
 8         audi a4 quattro   1.8  1999     4 manual(m5)     4    18    26
 9         audi a4 quattro   1.8  1999     4   auto(l5)     4    16    25
10         audi a4 quattro   2.0  2008     4 manual(m6)     4    20    28
# ... with 224 more rows, and 2 more variables: fl <chr>, class <chr>
```

==

Among the variables in this set are:

1. `displ`, a car's engine size in liters
2. `hwy`, a car's fuel efficiency in miles per gallon.

Plotting with ggplot2

Let's look at `displ` on the x-axis and `hwy` on the y-axis

```
mpg <- ggplot2::mpg # assign the dataset to a variable.

ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy), size = 2) +
  # all of the text below is just tweaking it for readability
  # this happens automatically in RStudio but is needed for
  # presentations
  theme_minimal() +
  theme(axis.text.x=element_text(size = 14),
        axis.text.y=element_text(size = 14),
        axis.title.x=element_text(size = 16),
        axis.title.y=element_text(size = 16))
```

ggplot2 template

We can use the code we just graphed as a template for all of our work in ggplot2. It goes something like this:

```
ggplot(data=<DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

`<MAPPINGS>` are going to be our first component to focus on.

Aesthetic mappings

> "The greatest value of a picture is when it forces us to notice what we never expected to see." — John Tukey

The map we just looked at showed there was a relationship between engine size and fuel efficiency. But it didn't account for the class of the car, so SUVs looked the same as subcompact cars. Let's add another variable to our graph. We'll use color to denote the class of the car.

==

```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy, color = class), size = 2) +
    theme_minimal() +
  theme(axis.text.x=element_text(size = 14),
        axis.text.y=element_text(size = 14),
        axis.title.x=element_text(size = 16),
        axis.title.y=element_text(size = 16))
```

Facets

So, we've seen that the class of car influences where they are on the `displ` vs `mpg` graph. How do the cars distribute within each of their respective classes?

This is where one of the powerful features of `ggplot` comes into play: Faceting.

==

Breaking up the graph by vehicle `class`

```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy), size=2) +
  theme(axis.text.x=element_text(size = 12),
        axis.text.y=element_text(size = 12),
        axis.title.x=element_text(size = 14),
        axis.title.y=element_text(size = 14)) +
  facet_wrap(~class, nrow = 2)
```

==

Faceting using two variables: `drv`, the drivetrain of the vehicle, and `cyl`, the number of cylinders in the engine.

```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy), size=2) +
  theme(axis.text.x=element_text(size = 12),
        axis.text.y=element_text(size = 12),
        axis.title.x=element_text(size = 14),
        axis.title.y=element_text(size = 14)) +
  facet_grid(drv ~ cyl)
```

Geometric objects (geoms)

Now let's look at `geom` instead of `aes`. Geoms are geometric objects, which tell ggplot2 what kind of graph to make. We've used `geom_point()` to make a scatter plot.

But we could display the same data differently…

Different geoms

===

```
# left
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy), size=2)

# right
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

===

We can even combine these two different **geoms** into a single plot. `ggplot` is built on the idea that you can layer information into a single graph.

```
ggplot(data=mpg) +
  geom_point(mapping = aes(x=displ, y=hwy), size=2) +
  geom_smooth(mapping = aes(x=displ, y= hwy)) +
  theme(axis.text.x=element_text(size = 12),
        axis.text.y=element_text(size = 12),
        axis.title.x=element_text(size = 14),
        axis.title.y=element_text(size = 14))
```

==

The previous plot introduced duplications in our code. We had to specify the aesthetic mappings for each geom, which could lead to typos. When you put mappings inside of the geom, it applies them to *that layer only*. To fix that, you can do this:

```
ggplot(data=mpg, mapping = aes(x=displ, y=hwy)) + # aes() is in the global environment
  geom_point(size=2) +
  geom_smooth() +
  theme(axis.text.x=element_text(size = 12),
        axis.text.y=element_text(size = 12),
        axis.title.x=element_text(size = 14),
        axis.title.y=element_text(size = 14))
```

=== > "Too often diagrams rely solely on one type of data or stay at one level of analysis." - Edward Tufte

Then, you can apply layer-specific aesthetics to the layer you want:

```
ggplot(data=mpg, mapping = aes(x=displ, y=hwy)) + # aes(x, y) are global (inherited by all geoms)
  geom_point(mapping = aes(color = class), size=2) + # aes(color) is only part of geom_point here
  geom_smooth(data=filter(mpg, class == "subcompact"), se = FALSE)
```

geom_bar

type:section

This section looks at ggplot's `geom_bar()` geom and related aesthetics.

==

We're going to use a new dataset. `diamonds` is a dataset in `ggplot2` that contains info on ~54,000 diamonds, including `price`, `carat`, `color`, `clarity` and `cut`.

```
diamonds
```

```
# A tibble: 53,940 x 10
   carat       cut color clarity depth table price     x     y     z
   <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
 1  0.23     Ideal     E     SI2  61.5    55   326  3.95  3.98  2.43
 2  0.21   Premium     E     SI1  59.8    61   326  3.89  3.84  2.31
 3  0.23      Good     E     VS1  56.9    65   327  4.05  4.07  2.31
 4  0.29   Premium     I     VS2  62.4    58   334  4.20  4.23  2.63
 5  0.31      Good     J     SI2  63.3    58   335  4.34  4.35  2.75
 6  0.24 Very Good     J    VVS2  62.8    57   336  3.94  3.96  2.48
 7  0.24 Very Good     I    VVS1  62.3    57   336  3.95  3.98  2.47
 8  0.26 Very Good     H     SI1  61.9    55   337  4.07  4.11  2.53
```

```
 9  0.22      Fair     E     VS2  65.1    61   337  3.87  3.78  2.49
10  0.23 Very Good     H     VS1  59.4    61   338  4.00  4.05  2.39
# ... with 53,930 more rows
```

===

```
ggplot(data=diamonds) +
  geom_bar(mapping = aes(x=cut)) +
  theme(axis.text.x=element_text(size = 12),
        axis.text.y=element_text(size = 12),
        axis.title.x=element_text(size = 14),
        axis.title.y=element_text(size = 14))
```

===

Bar charts introduce a new aesthetic: `fill`. You can still use `color` but that only affects the outline. `fill` will change the interior of the bar.

```
ggplot(data=diamonds) +
  geom_bar(mapping = aes(x=cut, fill=clarity)) +
  theme(axis.text.x=element_text(size = 12),
        axis.text.y=element_text(size = 12),
        axis.title.x=element_text(size = 14),
        axis.title.y=element_text(size = 14))
```

==

We can use `position` to change how the data are displayed. `fill` creates a 100% area bar chart, useful for comparing proportions.

```
ggplot(data=diamonds) +
  geom_bar(mapping = aes(x=cut, fill=clarity), position = "fill") +
  theme(axis.text.x=element_text(size = 12),
        axis.text.y=element_text(size = 12),
        axis.title.x=element_text(size = 14),
        axis.title.y=element_text(size = 14))
```

==

`dodge` places overlapping objects directly beside one another.

```
ggplot(data=diamonds) +
  geom_bar(mapping = aes(x=cut, fill=clarity), position = "dodge") +
  theme(axis.text.x=element_text(size = 12),
        axis.text.y=element_text(size = 12),
        axis.title.x=element_text(size = 14),
        axis.title.y=element_text(size = 14))
```

ggthemes

Creating your own theme is time consuming up-front. `ggthemes` can help.

```
library(ggthemes)
ggplot(data=diamonds) +
  geom_bar(mapping = aes(x=cut, fill=clarity), position = "dodge") +
  theme_minimal()
```

ggThemeAssist

A `shiny` based application for those who like GUIs.

```
library(ggThemeAssist)

p <- ggplot(data=diamonds) +
  geom_bar(mapping = aes(x=cut, fill=clarity), position = "dodge")

ggThemeAssist::ggThemeAssistGadget(p)
```

For more information

Tutorials:

Introducing R to Excel Users (great blog post w/ examples) https://www.jessesadler.com/post/excel-vs-r/

Nice but dated (2012) tutorial on ggplot2 http://www.ling.upenn.edu/~joseff/avml2012/

Another nice ggplot2 tutorial (also dated) http://tutorials.iq.harvard.edu/R/Rgraphics/Rgraphics.html

Guided Learning:

DataCamp courses are free to start, but cost money to access later chapters. https://www.datacamp.com/courses/data-visualization-with-ggplot2-1

`swirl` package: "Learn R in R"

```
install.packages("swirl") # install package
swirl() # to start the program from the command line
```

Where to get help?

Asking questions:

- StackOverflow
- ggplot2 Google Group

Or, you know… me. (Email: aaron.c.cochran@state.or.us, Phone: 503-945-6867)

Excellent blogs: * Simply Statistics * Revolution Analytics * R-bloggers

Social Media: * RStudio Tips Twitter * Rbloggers Twitter * RTips Twitter

For fun: * RCatLadies Twitter (Gender inclusive!)

Lab 1

type:section

Time for some exercises.

Labs are available at https://github.com/DHS-OEDA/r_training/