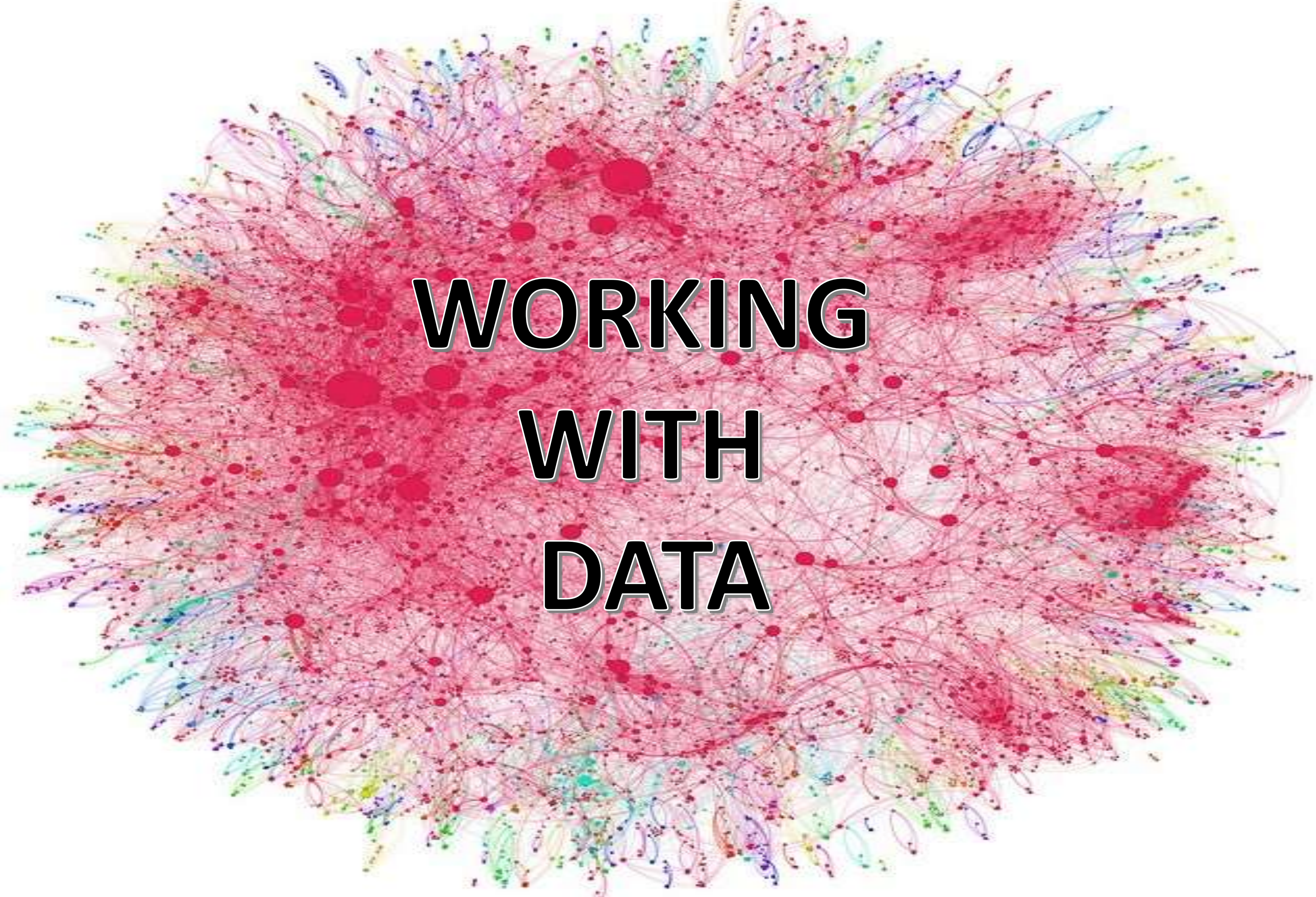


WORKING WITH DATA

UNIVERSITY of HOUSTON
DIVISION OF RESEARCH
HPE DATA SCIENCE INSTITUTE

Working with Data - Data Wrangling

- Variable Types & Data Structures
- Import, Dealing with Missing Data
- Transformation, Subsetting, Merging & Reshaping
- Data Cleaning
- Data Export

Variables in R Summary

- character: "treatment", "123", 'A', "A"
- numeric: 23.44, 120, NaN, Inf
- integer: 4L, 1123L
- logical: TRUE, FALSE, NA
- factor: factor("Hello"), factor(8)
  (see next slide)

```
> class("hello")
[1] "character"

> class(3.844)
[1] "numeric"

> class(77L)
[1] "integer"

> class(factor("yes"))
[1] "factor"

> class(TRUE)
[1] "logical"
```

Factors (very important!)

- categorical variables for when we would prefer numeric values with associated labels, they don't have to be labeled.

- most important uses of factors: statistical modeling; since categorical variables enter into statistical models differently than continuous variables, storing data as factors insures that the modeling functions will treat such data correctly.

- Example:

```
> a <- factor (c("a", "b", "c", "b", "c", "b", "a", "c", "c")) # create the factor

> a                         # Print the new variable

[1] a b c b c b a c c       # You can tell those are not stored as character: no quotes

Levels: a b c               # Also the levels print out

> levels(a)                 # You can get the set of levels separately
```

Type conversion

```
> as.character(2016)

> [1] "2016"

> as.numeric(TRUE)

> [1] 1

> as.integer(99)

> [1] 99

> as.factor("something")

> [1] something Levels: something
> as.logical(0)

> [1] FALSE
```

How to deal with dates & times

- package *lubridate*

```
# Load the lubridate package
> library(lubridate)

# Experiment with basic lubridate functions
> ymd("2015-08-25")
[1] "2015-08-25 UTC"      year-month-day

> ymd("2015 August 25")
[1] "2015-08-25 UTC"      year-month-day

> mdy("August 25, 2015")
[1] "2015-08-25 UTC"      month-day-year

> hms("13:33:09")
[1] "13H 33M 9S"      hour-minute-second

> ymd_hms("2015/08/25 13.33.09")
[1] "2015-08-25 13:33:09 UTC"      year-month-day hour-minute-second
```

Practice

```
# Load the lubridate package

>

# create a character type object ("17 Sep 2015")
and name it dob

>

# Coerce dob to a date and store as object mydate

>
```

Operators

Arithmetic Operators	+,-,*,/,^,%%
Relational Operators	>,<,==,!=
Logical Operators	&,|,!
Assignment Operators	<− or = or ->
Miscellaneous Operators	:, %in%

Practice

```
> v <- c( 2,5.5,6); t <- c(8, 3, 4)

> v^t

> v%%t

> v1 <- c(3,1,TRUE,2+3i);
c(3,1,TRUE,2+3i) -> v2;
v3 = c(3,1,TRUE,2+3i)

> v|t; v||t

> v <- 2:8
```

R Data Structures Summary

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data Frame Tibble
nd	Array	

R Data Structures

- **Vectors**

```
> a <- c(1,2,5.3,6,-2,4) # numeric vector
> a
> b <- c("one","two","three") # character vector
> b
> c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE) #logical vector
> (c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE)) #logical vector
```

- **Matrices** (All columns in a matrix must have the same mode(numeric, character, etc.) and the same length)

```
>y <- matrix(1:20, nrow=5, ncol=4) # generates 5 x 4 numeric
matrix

> cells <- c(1,26,24,68)
> rnames <- c("R1", "R2")
> cnames <- c("C1", "C2")
> mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=TRUE,
  dimnames=list(rnames, cnames))
```

Practice

Create a vector of red, green and yellow

>

Create the magic matrix ->

4	9	2
3	5	7
8	1	6

>

Create a 3*3 identity matrix

>

R Data Structures cont.

- **Arrays** are similar to matrices but can have more than two dimensions

```
> a <- array(c(“green”,”yellow”),dim = c(3,3,2))
```

- **Data Frames** are more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.)

  Are the most commonly used data structure in R

```
> d <- c(1,2,3,4)
> e <- c("red", "white", "red", NA)
> f <- c(TRUE,TRUE,TRUE,FALSE)
> mydata <- data.frame(d,e,f)
> mydata
> names(mydata) <- c("ID","Color","Passed") # variable names
```

Practice

Create a 3*3*3 array full of ones

```
>
```

Create a data frame with 10 rows and 3 columns, first column with all 1, second column with numbers 1 to 10 and third column with a letter randomly selected from A,B,C (hint: use code below for third column)

```
> L3 <- LETTERS[1:3]; fac <- sample(L3, 10, replace = TRUE)

>
```

Tibbles

- are data frames, but they tweak some older behaviors to make life a little easier
  - more elegant printing of data
- it never changes the type of the inputs (e.g. it never converts strings to factors!), it never changes the names of variables, and it never creates row names.
- can have column names that are not valid R variable names, aka non-syntactic names.
  (A syntactically valid name in R consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number. Names such as ".2way" are not valid, and neither are the reserved words, like "for")

Creating Data - *sampling* functions

- we will simply create some data using sampling functions

```
>x <- sample(c('Heads', 'Tails',
'Edge', 'Blows Up'), 5,
replace=T,  prob=c(.45, .45, .05,
.05))

> x2 <- rbinom(5, 1, .5)
> x3 <- rnorm(50, mean=50, sd=10)

> set.seed(Sys.time())
```

Creating Data - Tibbles

```
> library(tidyverse)

> as_tibble(iris)

> tibble(

  x = 1:5,

  y = 1,

  z = x ^ 2 + y

)
```

```
#> # A tibble: 5 × 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
```

```
#> # A tibble: 150 × 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl>  <fctr>
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
#> # ... with 144 more rows
```

Exercise

- How can you tell if an object is a tibble?

- Compare and contrast the following operations on a data.frame and equivalent tibble. What is different?

```
> df <- data.frame(abc = 1, xyz = "a")

> df$xyz

> df[, "xyz"]

> df[, c("abc", "xyz")]
```

Importing Data

✓ R can read data from files

- Very important concept: **Working Directory** (this is where R will read data from by default)

```
> getwd()              # get current working directory

> setwd("<new path>")  # set working directory
```

Note that the forward slash should be used as the path separator even on Windows platform > setwd(“C:/ MyDoc")

File Import - Data Tables

Table File

- A data table can reside in a text file. The cells inside the table are separated by blank characters. Here is an example of a table with 5 rows and 3 columns. <u>Our example files are all to be found in the RExamples folder. Please download it before the next exercise.</u>

```
>mydata <- read.table("mydata.txt") # read text file
```

100	a1	b1
200	a2	b2
300	a3	b3
400	a4	b4
500	a5	b5

File Import - csv

CSV File

- Each cell inside is separated by a special character, which usually is a comma, although other characters can be used as well. The first row of the data file should contain the column names instead of the actual data.

```
> mydata = read.csv("mydata.csv")  # read csv file
```

```
Col1,Col2,Col3
100,a1,b1
200,a2,b2
300,a3,b3
```

- more import functions - http://www.r-tutor.com/r-introduction/data-frame/data-import

Import - CSV Example

The behavior of the different import functions varies slightly.

```
>data<-
read.csv(“household_power_consumption.txt",
sep=";", header = FALSE,
stringsAsFactors=FALSE, na.strings = "?",
skip=66637 , nrows=2880)
```

#set the column names

```
>colnames(data) <-
names(read.csv("household_power_consumptio
n
.txt", sep=";", nrows=1))
```

File Import - Excel file

- Quite frequently, the sample data is in Excel format, and needs to be imported into R prior to use. For this, we can use the functions from the *readxl* package. It reads from an Excel spreadsheet and returns a data frame.

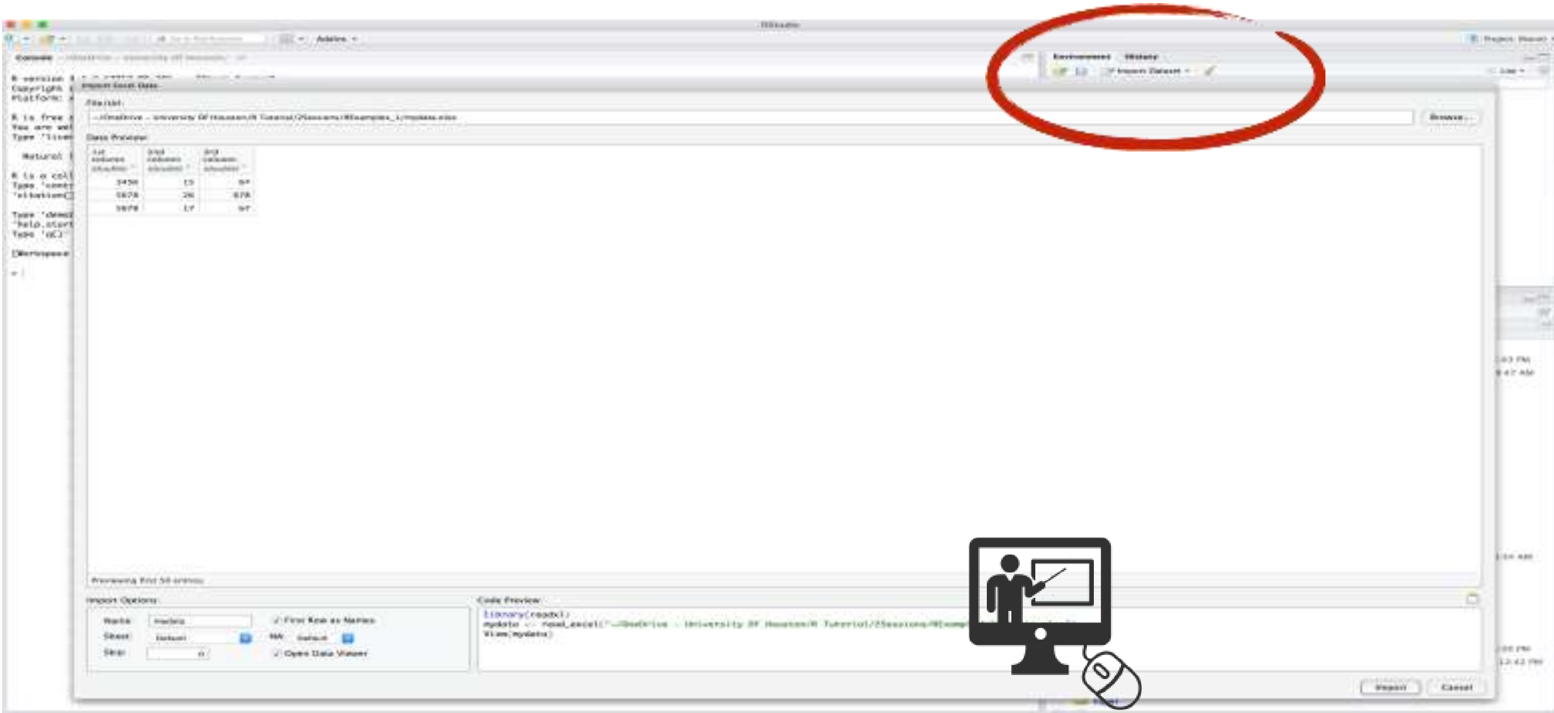
```
> library(readxl)   # load readxl package

>mydata <- read_xls("mydata.xls")    # read
from first sheet

> mydata <- read_excel("mydata.xlsx")
```

- Recommendation when issues occur: Store Excel file as tab separated file and use RStudio "Import" function.

Using Studio for import



Working with Data - Helpful commands

Get to know your data ...

```
> ?mtcars              # General info about data set

> head(mtcars)         # First couple of lines

                       # Shows that the data is a data frame: A rectangular structure
> str(mtcars)          # Each column has same type, but different
                       # columns may have different types

> names(mtcars)        # List the column names

> summary(mtcars)      # summary statistics
```

Dealing with Missing Values

- In R, missing values are represented by the symbol **NA** (not available). Impossible values (e.g., dividing by zero) are represented by the symbol **NaN** (not a number). Unlike SAS, R uses the same symbol for character and numeric data.

- Testing for missing values (NA == NA # Is NA!)

```
> is.na(x) # returns TRUE of x is missing
> y <- c(1,2,3,NA)
> is.na(y) # returns a vector (F F F T)
```

- Recoding Values to Missing (if your data uses a different code for missing values)

```
# recode 99 to missing for variable Col1
# select rows where Col1 is 100 and recode column Col1
> mydata$Col1[mydata$Col1==100] <- NA
```

Dealing with Missing Values

- Counting missing values

```
> x <- c(1, 2, NA, 4)

>sum(is.na(x)) # sums up the missing values in a column

> 1
```

- Which one is NA?

```
> which(is.na(x))

> 3
```

Dealing with Missing Values

- Excluding Missing Values from Analyses is often necessary since the default is to propagate missing values. Many functions have *na.rm* argument to remove them

```
> x <- c(1,2,NA,3)
> mean(x) # returns NA
> mean(x, na.rm=TRUE) # returns 2
```

- The function *complete.cases()* returns a logical vector indicating which cases are complete.

```
# list rows of data that have missing values
> mydata[!complete.cases(mydata),]
```

- The function *na.omit()* returns the object with listwise deletion of missing values.

```
# create new dataset without missing data
> newdata <- na.omit(mydata)
```

Advanced Handling of Missing Data

- Most modeling functions in R offer options for dealing with missing values. You can go beyond pairwise and listwise deletion of missing values through methods such as multiple imputation. Good implementations that can be accessed through R include:

  - Amelia II (http://gking.harvard.edu/amelia/)

  - Mice (https://www.rdocumentation.org/packages/mice/versions/2.25/topics/mice)

  - mitools (http://cran.us.r-project.org/web/packages/mitools/index.html)

The Data Exploration Process

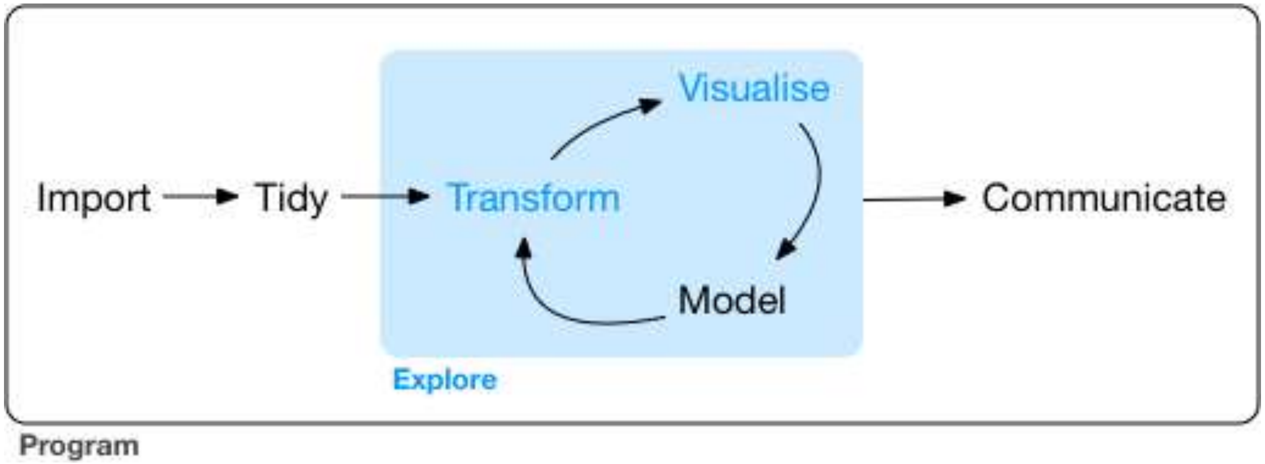


Image courtesy of: R for Data Science
http://r4ds.had.co.nz/explore-intro.html

Data Transformation with *dplyr*

- following *tidy verse* - introduction of *dplyr* package to work with most common data manipulation challenges:
  - Pick observations by their values (*filter()*).
  - Reorder the rows (*arrange()*).
  - Pick variables by their names (*select()*).
  - Create new variables with functions of existing variables (*mutate()*).
  - Collapse many values down to a single summary (*summarise()*).
- These can all be used in conjunction with *group_by*() which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a <u>language of data manipulation</u>.
- All verbs (the 6 functions) work similarly:
  - The first argument is a data frame.
  - The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
  - The result is a new data frame.

https://www.tidyverse.org/

Data Transformation - *filter()*

```
> library(nycflights13)
> library(tidyverse)

> flights
```

- Filter rows

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

> jan1 <- filter(flights, month == 1, day == 1)
```

filter() excludes FALSE and NA

- Comparisons

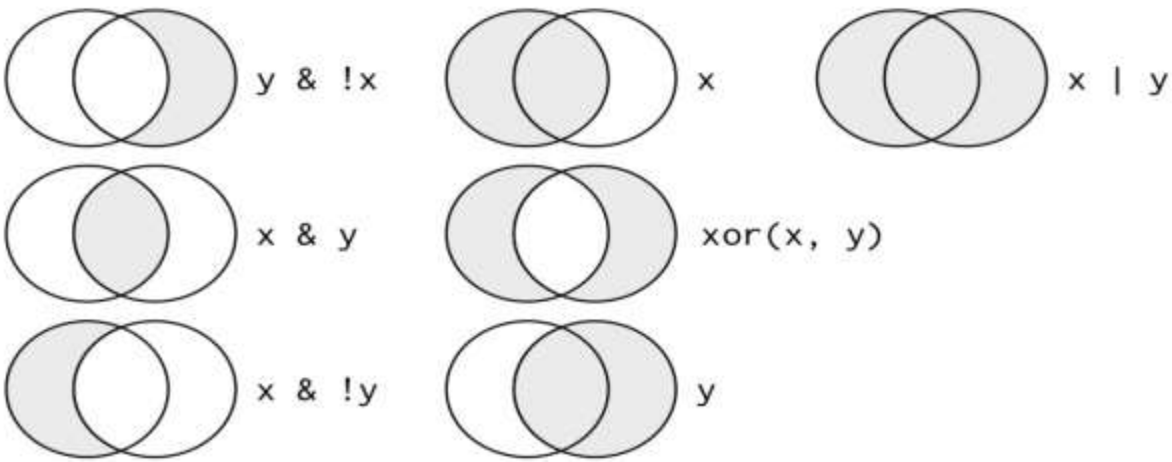
```
> filter(flights, month = 1)

 #> Error: filter() takes unnamed arguments. Do
you  need `==`?
```

Data Transformation - *filter()*

- Using the logical operators & (AND), ! (NOT), |(OR)

```
> filter(flights, month == 11 | month == 12)
```



Complete set of boolean operations. `x` is the left-hand circle, `y` is the right-hand circle, and the shaded region show which parts each operator selects.

Image courtesy of:
R for Data Science, http://r4ds.had.co.nz/transform.html

Data Transformation - helper functions

- Useful shortcut *x %in% y*

```
> nov_dec <- filter(flights, month
%in% c(11, 12))
```

- another helper is *between()*

```
>summer <- filter(flights,
between(month, 6, 8))
```

Practice

- Using the nycflights13 dataset, find all flights that
  - Had an arrival delay of two or more hours

```
>
```

  - Flew to Houston (IAH or HOU)

```
>
```

Data Transformation - *arrange()*

- Arrange rows to change order

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

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

- missing values always show up at the end

```
> df <- tibble(x = c(5, 2, NA))

> arrange(df, x)

> arrange(df, desc(x))
```

Data Transformation - *select()*

- Select columns

```
# Select columns by name
> select(flights, year, month, day)
# Select all columns between year and day (inclusive)
> select(flights, year:day)
# Select all columns except those from year to day
(inclusive)
> select(flights, -(year:day))
```

- helper functions: *starts_with(“abc”)* matches names that begin with “abc”, *ends_with(“xyz”)* matches names that end with “xyz”, *contains(“ijk")*:matches names that contain “ijk”, *matches("(.)\\1")* selects variables that match a regular expression, *num_range(“x", 1:3)* matches x1, x2 and x3.

Data Transformation - mutate()

- Add new variables

```
>   flights_sml <-
    select(flights,  year:day,
    ends_with("delay"),
    distance,
    air_time
  )
> mutate(flights_sml,
    gain = arr_delay - dep_delay,
    speed = distance / air_time * 60
  )
```

```
> mutate(flights_sml,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
> transmute(flights,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```

Data Transformation - *mutate()*

- There are many functions for creating new variables that you can use with mutate().
  - Arithmetic operators: +, -, *, /, ^.
  - Arithmetic operators are also useful in conjunction with the aggregate functions you'll learn about later. For example, x / sum(x) calculates the proportion of a total, and y - mean(y) computes the difference from the mean.
  - Modular arithmetic: *%/%* (integer division) and *%%* (remainder), where x == y * (x %/% y) + (x%% y).
  - Logs: *log(), log2(), log10().*
  - Offsets: *lead()* and *lag()* allow you to refer to leading or lagging values.
  - Cumulative and rolling aggregates: R provides functions for running sums, products, mins and maxes: cumsum(), cumprod(), cummin(), cummax(); and dplyr provides cummean() for cumulative means. If you need rolling aggregates (i.e. a sum computed over a rolling window), try the RcppRoll package.
  - Logical comparisons, <, <=, >, >=, !=, which you learned about earlier.
  - Ranking: start with *min_rank()*.

Data Transformation - *summarise()*

- Collapse a data frame into single row

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

- pair with *group_by()*

```
> by_day <- group_by(flights, year, month, day)

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

Data Transformation - *Combinations*

- combine using "the pipe"
- Example: There are three steps to prepare this data:
  1. Group flights by destination.
  2. Summarize to compute distance, average delay, and number of flights.
  3. Filter to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport.

```
> delays <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")
```

Data Transformation - *pipe*

- more function to summarize
  - *count()*
  - *mean() median()*

```
> not_cancelled <- flights %>%
  group_by(year, month, day) %>%
  summarise(
    avg_delay1 = mean(arr_delay),
    avg_delay2 = mean(arr_delay[arr_delay > 0]) # the average positive delay
  )
```

  - *sd(x), IQR(x), mad(x)*
  - Measures of rank: min(x), quantile(x, 0.25), max(x)
  - Measures of position: first(x), nth(x, 2), last(x)

Data Transformation - more grouping

- Grouping by multiple variables

```
> daily <- group_by(flights, year, month, day)

(per_day   <- summarise(daily, flights = n(), sd(x), IQR(x),
mad(x)))

(per_month <- summarise(per_day, flights = sum(flights)))

(per_year  <- summarise(per_month, flights = sum(flights)))
```

- Ungrouping

```
> daily %>%

  ungroup() %>%               # no longer grouped by date

  summarise(flights = n())    # all flights
```

Working with Data - Subsetting

- Mastering indexing/subsetting is critical for efficient R programming, e.g.

```
x[,4] # 4th column of matrix/data frame
x[3,] # 3rd row of matrix/data frame
x[2:4,1:3] # rows 2,3,4 of columns 1,2,3
```

```
> mystates <- data.frame(state.x77)
> mystates[1:5, ]
> mystates[,'Area']
> mysubset = subset(mystates, state.region == “South")
> mysubset = mystates[state.region == "South", ]
> mysubset = mystates[, c(1:2, 7:8)]
>mysubset = mystates[, c("Population", "Income",
"Frost", “Area”)]
```

get any States starting with 'I' and ending with 'a' using regular # expressions

```
> mysubset = state2[grep("^I.*a$", rownames(state2)), ]
```

Practice

Using the state.x77 data find the state with largest area

>

Working with Data - Merging

- important: make sure you know what you want to merge, check data types, length of variables etc.

- Example (we first create the data for the example):

```
> mydat <- data.frame(id = factor(1:12), group = factor(rep(1:2, e
= 3)))

> x = rnorm(12)
> y = sample(70:100, 12)
> x2 = runif(12)
```

- add columns

```
> mydat$grade = y #add y via extract operator
> df <- data.frame(id = mydat$id, y)
> mydat2 <- merge(mydat, df, by = "id", sort = F) #using merge
> mydat3 <- cbind(mydat, x) #using cbind
```

- add rows

```
>df <- data.frame(id = factor(13:24), group = factor(rep(1:2, e =
3)), grade = sample(y))
> mydat2 <- rbind(mydat, df)
```

Working with Data - Fixing Characters

- Replace the first occurrence of a pattern with `sub(pattern, replacement, x)` or replace all occurrences with `gsub(pattern, replacement, x)`.
  - pattern – A pattern to search for, which is assumed to be a regular expression. Use an additional argument fixed=TRUE to look for a pattern without using regular expressions.
  - replacement – A character string to replace the occurrence (or occurrences for gsub) of pattern.
  - x – A character vector to search for pattern. Each element will be searched separately.

```
> testString <- “this_is_a_test”

> sub(“_”, “ “, testString)

> gsub(“_”, “ “, testString)
```

More useful String functions

```
> library(stringr)
>nchar("Your Name") [1] 9
> substring("Your Name",1,4)
[1] "Your"
> paste("Your", "Name")
[1] "Your Name"
> paste0("Your", "Name")
[1] "YourName"
> str_trim("Your              ")
[1] "Your"
```

Data Export

- As for import there are endless export options
- Check the arguments in the documentation for special cases

```
>write.table(mydata, “c:/Users/[username]/mydata.txt”, sep=“\t")

>write.csv(mydata, file = “mydata.csv", row.names = FALSE, quote = FALSE)

> library(xlsx)

> write.xlsx(mydata, "c:/Users/[username]/mydata.xlsx")

>write.xlsx(x = mydata, file = “testexcelfile.xlsx", sheetName = "TestSheet", row.names = FALSE)
```

Swirl

- http://swirlstats.com/students.html