

**UT Southwestern**  
Medical Center  
Lyda Hill Department of Bioinformatics

BioHPC

---

# Data handling with R

Amit Amritkar

# Working with Data - Data Wrangling

---

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

# Using Rstudio on BioHPC

<https://portal.biohpc.swmed.edu/content/> (Use VPN)

→ Cloud Services → RStudio

→ BioHPC OnDemand → OnDemand RStudio

The screenshot shows the BioHPC User Portal website. The header includes the UT Southwestern Medical Center logo and the BioHPC name. A navigation menu contains links for Home, News, About, Status, Training, Guides, FAQs, Cloud Services, BioHPC OnDemand, Links, Software, and Careers. A central banner announces that Nucleus has 12 new nodes equipped with 4GPUs each, with full node specs: 2x Intel Xeon Gold 6240 Processor (CPU) 36 physical cores 387GB of RAM 4x NVIDIA Tesla V100. Below the banner, a welcome message is followed by system status: Nucleus Queue (25 jobs pending, 361 jobs running) and Nodes Available (17 GPU, 81 super). Quota Usage is shown for /home, /work, /project, and /archive. Recently Top Used Modules include java, git, pycharm-ce, geany, and mesa. A hiring notice is also present. At the bottom, there are sections for News & Updates and Open Support Tickets.

If using RStudio OnDemand:

```
Sys.setenv(https_proxy = "http://proxy.swmed.edu:3128")
```

```
Sys.setenv(http_proxy = "http://proxy.swmed.edu:3128")
```

# 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)
  - categorical variables



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

> class(3.844)
[1] "numeric"

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

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

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

# 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
```

# Operators

---

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

# 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))
```

## 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
```



# 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
```

## 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. The example files are all to be found in the biohpc\_r zip file. Please download it here:

<https://cloud.biohpc.swmed.edu/index.php/s/mn4reqkpQrMJEbj>

- ```
> 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 , nrow=2880)  
  
> colnames(data) <-  
names(read.csv("household_power_consumption.txt"  
, sep=";", nrow=1)) #set the column names
```



## 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 RStudio for import

A screenshot of the RStudio interface. The 'Import Excel Data' dialog box is open, showing options for file selection, data preview, and import settings. The 'Import Dataset' button in the top toolbar is circled in red. The console on the left shows R version 4.1.0 and some library loading commands. The code preview on the right shows the R code for reading an Excel file.

RStudio

File Edit Code View Plots Session Build Debug Profile Tools Help

Go to file/function Addins Project: (None)

Console Terminal x Jobs x Environment History Connections Tutorial

R 4.1.0 ~/ Import Dataset 168 MiB

Import Excel Data

File/URL:  Browse...

Data Preview:

Import Options:

Name:  Max Rows:   First Row as Names

Sheet:  Skip:   Open Data Viewer

Range:  NA:

Code Preview:

```
library(readxl)
dataset <- read_excel(NULL)
View(dataset)
```

? Reading Excel files using readxl

Import Cancel

# 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 if 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

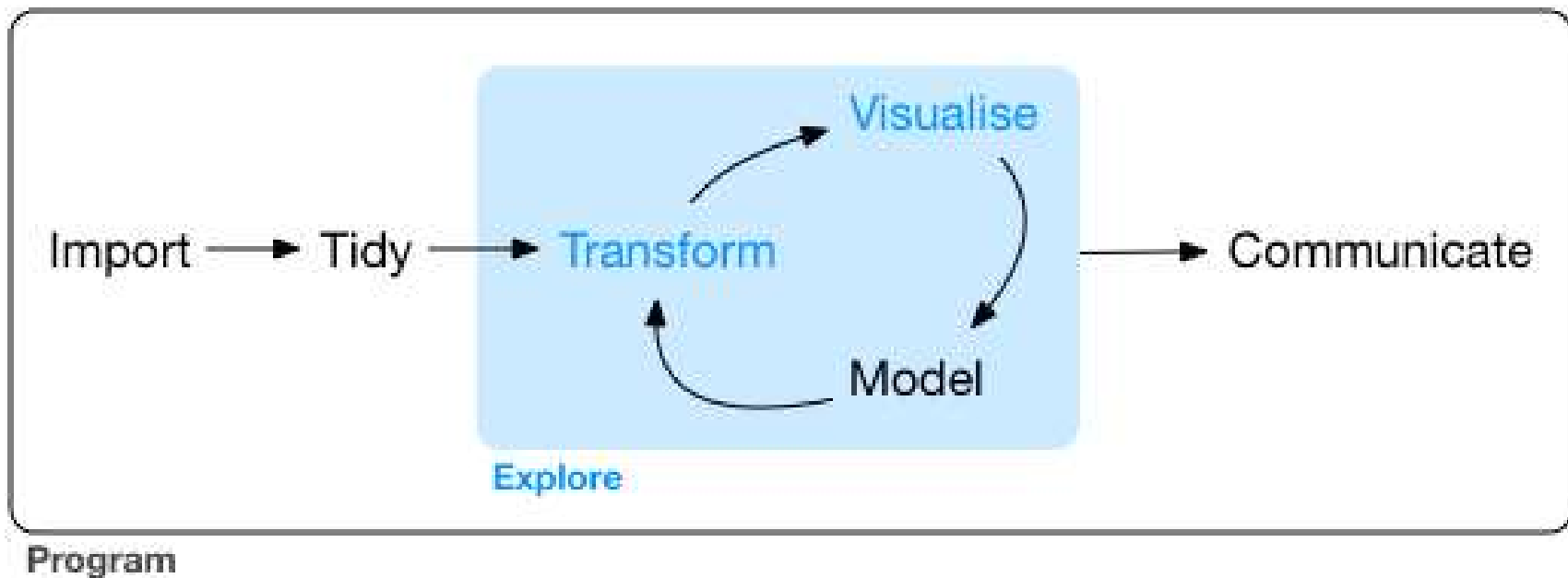


Image courtesy of: R for Data Science  
<http://r4ds.had.co.nz/explore-intro.html>

# 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 language of data manipulation.
- 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.

# Data Transformation - *filter()*

```
> library(nycflights13)
```

```
> library(tidyverse)
```

```
> flights
```

- Filter rows

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

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



- Comparisons

`filter()` excludes FALSE and NA

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

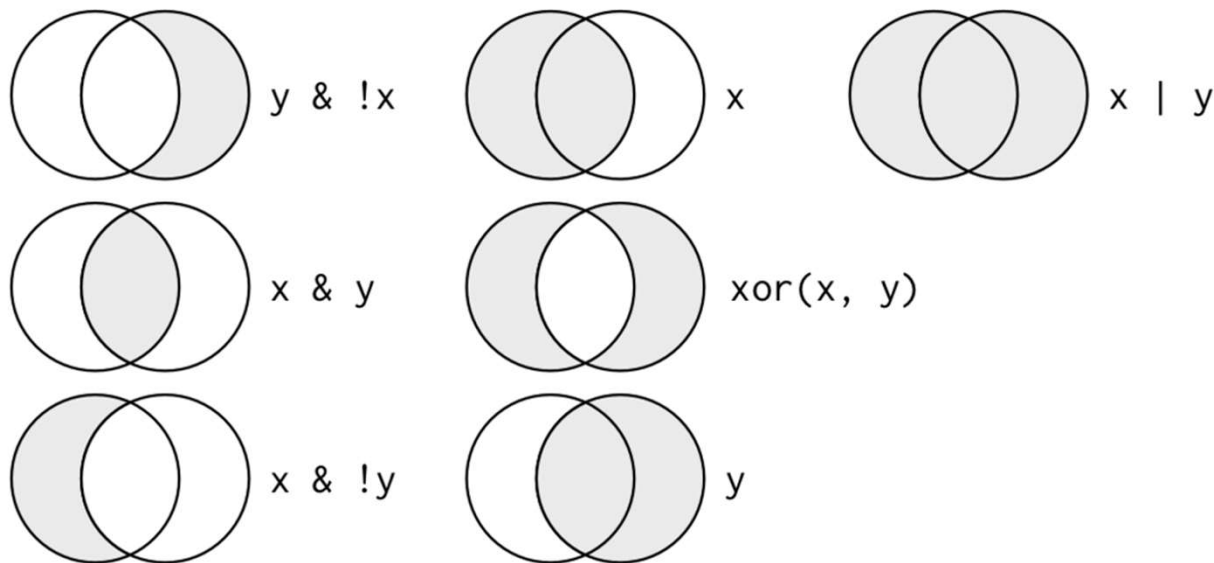
```
#> Error: filter() takes unnamed arguments. Do you need `==`?
```

# 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.

# 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))
```

# 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 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>