Applied Data Science: Lecture 1

David Shilane

What This Course Is All About

"When I was your age…"

Course Details

- STAT 4243/5243
- Instructor: David Shilane (david.shilane@columbia.edu)
- Weekly Lectures: Thursdays 1:10-3:40
- Homeworks: 4 assignments spread throughout the semester
- Midterm Project: In groups
- Final Project: In groups
- Grades: 40% Homework, 20% Midterm Project, 40% Final Project

No Unauthorized Distribution of Content

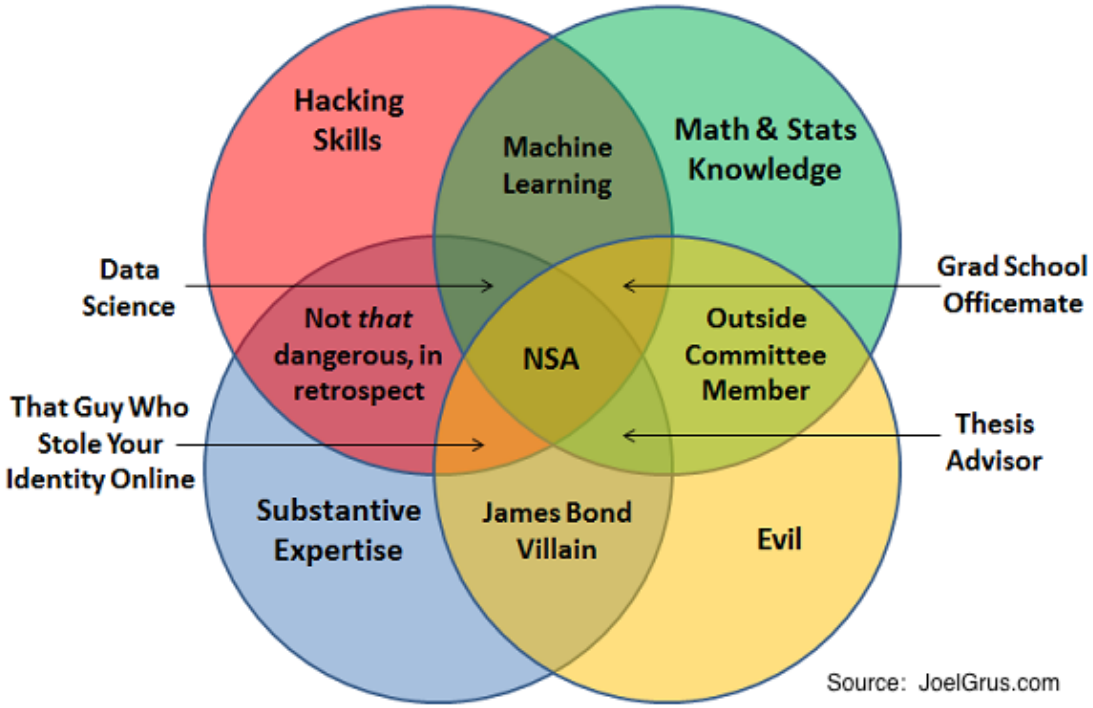
- Please do not post electronic copies of any materials you receive in this course.
- Sharing these files without prior written consent from the instructor is prohibited.

What We'll Focus On

- R, R, R.
- Becoming better Data Scientists
- Learning how to be practical… and productive!
- Case studies from different industries
  - *Education*
  - *Health Care*
  - *Marketing*
  - *Social Media*
  - *Online Business*

Pillars of Data Science

But There's Usually More To It

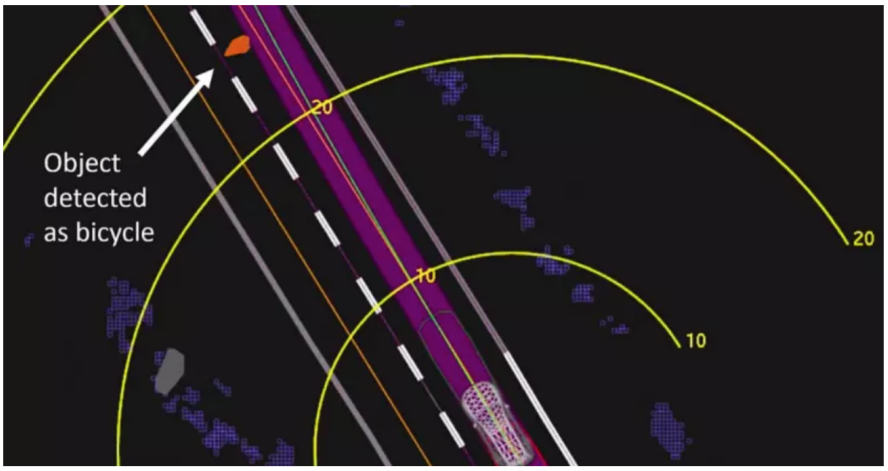


Ethics Are Paramount

https://usa.streetsblog.org/2018/05/24/how-ubers-self-driving-system-failed-to-brake-and-avoid-killing-elaine-herzberg/

How Uber's Self-Driving System Failed to Brake and Avoid Killing Elaine Herzberg

By Angie Schmitt | May 24, 2018 | 27

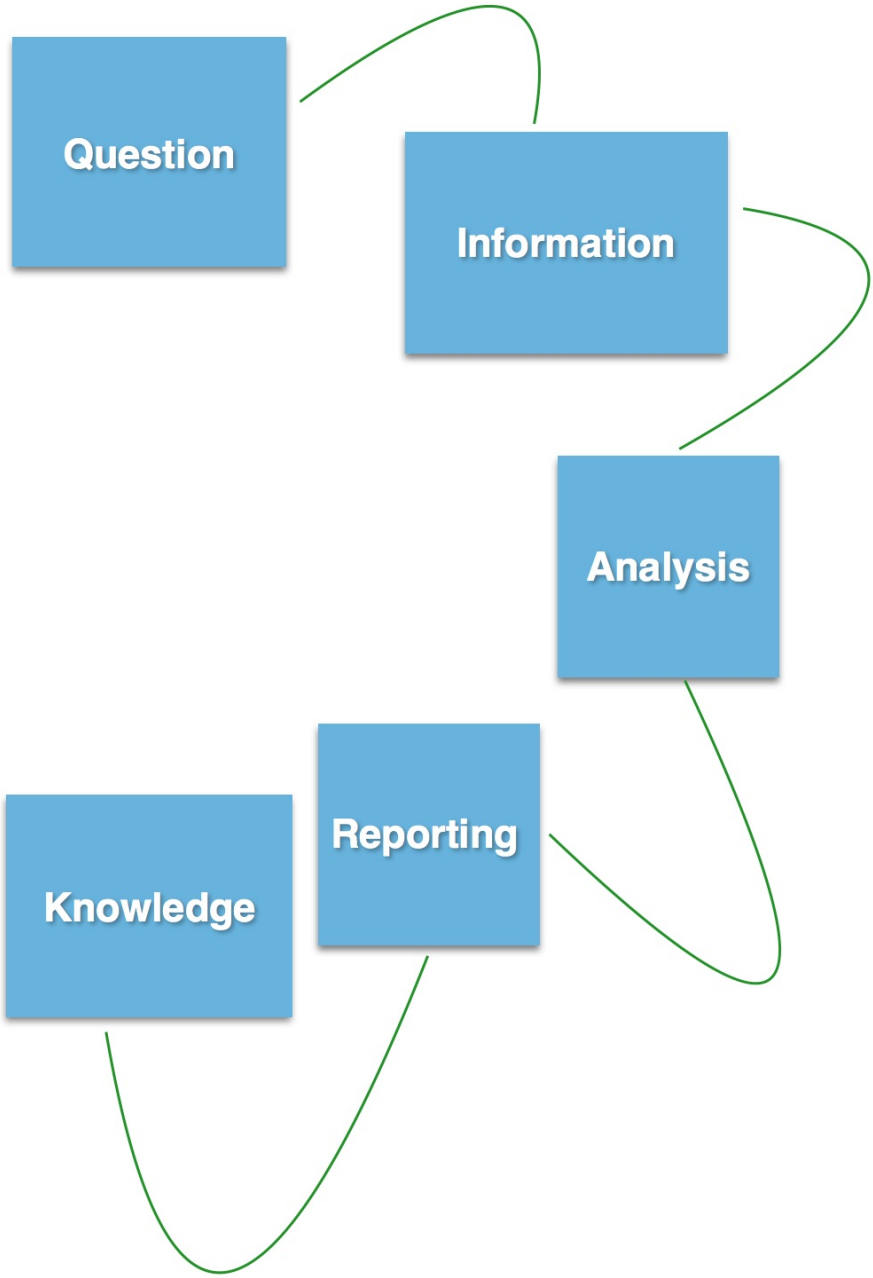


The self-driving system detected Elaine Herzberg six seconds before impact, but Uber had tuned the emergency braking feature to be too insensitive to respond in time. Image: NTSB

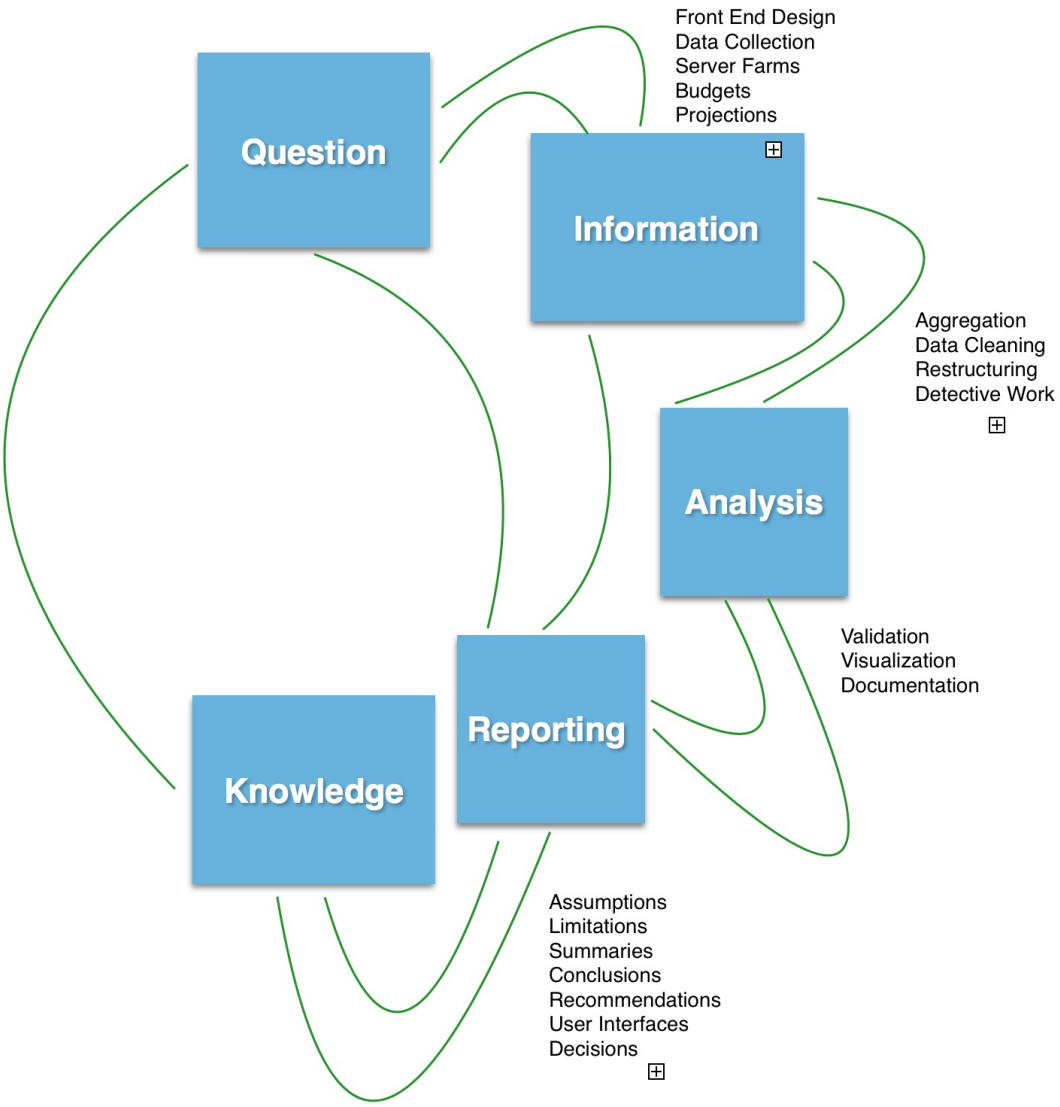
"The more cautiously a car's software is programmed, the more often it will slam on its brakes unnecessarily. That will produce a safer ride but also one that's not as comfortable for passengers."

In your work, you may be called upon to make difficult choices with trade-offs. Maintaining a strong ethical framework is an important component of making your choice.

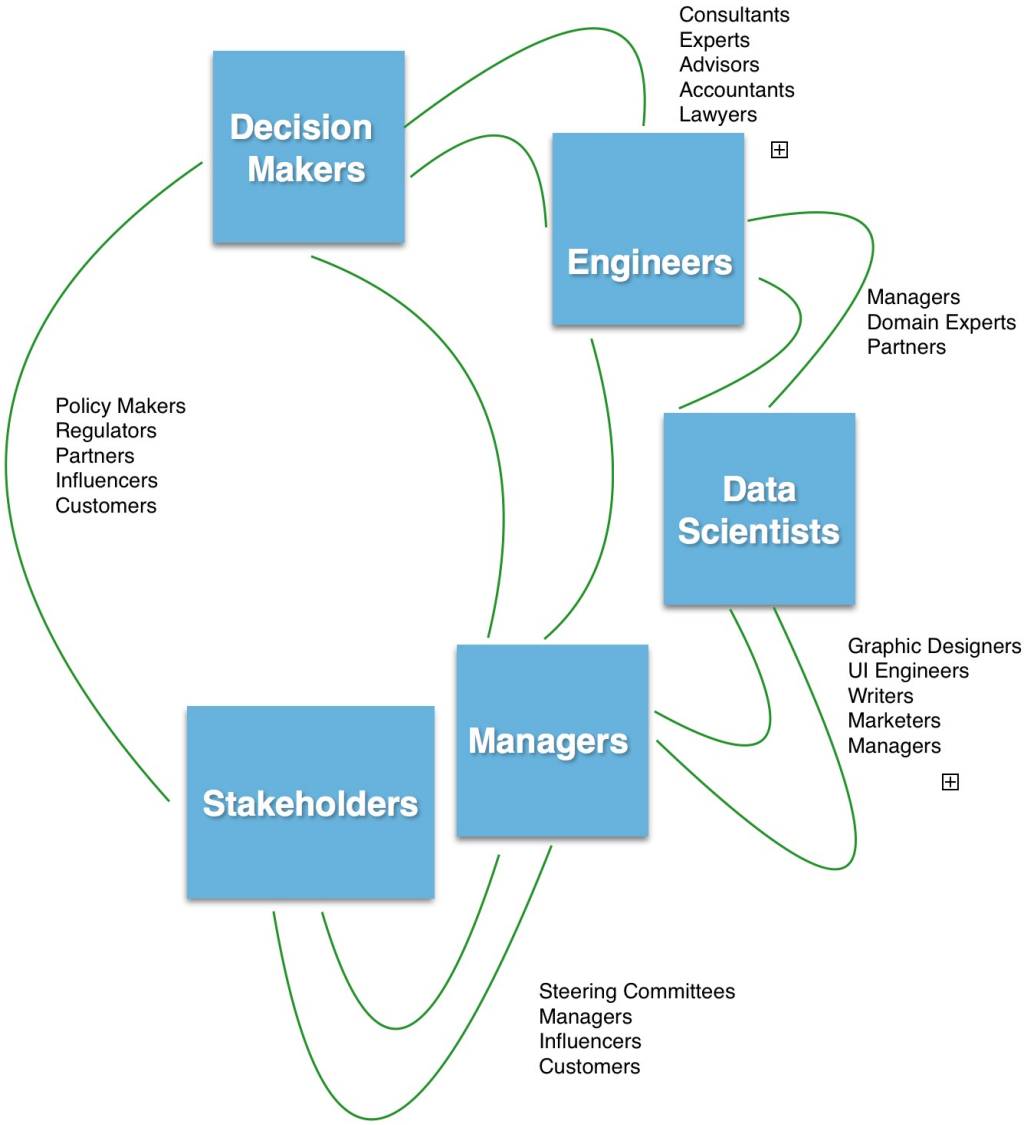
Seeking Knowledge



What It's Really Like



And Don’t Forget About The People



And What Is Data Engineering?

Data Engineering is anything upstream of analysis that a Data Scientist:

- Doesn't want to do… and
- Can convince a Data Engineer to do instead.

Enjoy the negotiation!

Not Just Engineers

- The teams you work on could take many forms.
- Data might be collected by people (Census, medical case notes, user-supplied).
- Understanding the process can mean understanding the practices of the people involved as much as the logic of the technical systems.

Aren't Data Scientists Really Just Statisticians?

Yes, maybe, I don't know, but probably yes!

Many fields (medicine, economics, psychology, business) have co-opted statistics to solve some of their problems. In some cases, they have tried to rename the discipline (like biostatistics, econometrics, psychometrics, business analytics, etc.).

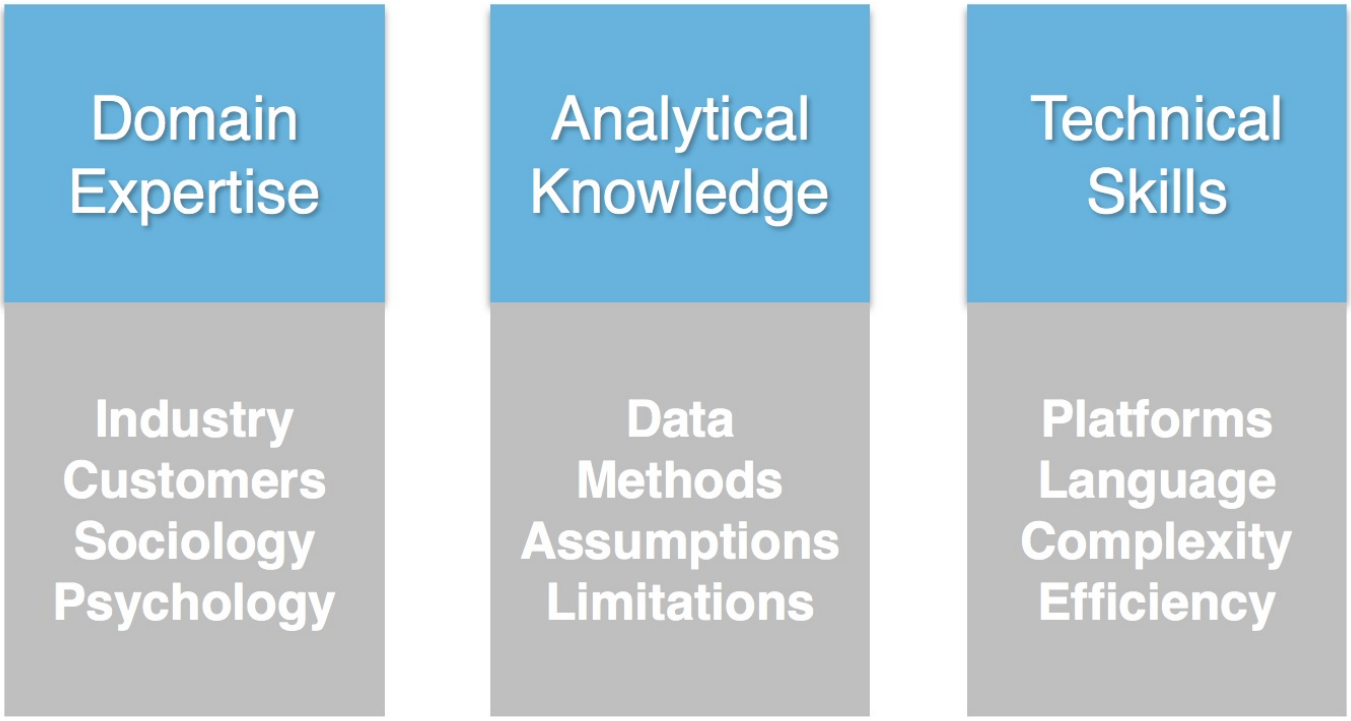
Unlike the other fields, computer science has sought this kind of rebranding at least 3 times – with **artificial intelligence**, **machine learning**, and now **data science**, a term that was barely known before about 2012. Through brute force and the power of marketing, data science is increasingly ascendant. However, its analytical goals, which may require enormous computational resources, are very much in the realm and tradition of statistics.

But If We Have to Make a Distinction

- Data scientists are more likely to have studied science or engineering, while statisticians traditioinally have a stronger background in mathematics.
- Data scientists in practice have a greater expectation for doing some of the engineering work prior to the analysis. There is less of an expectation of starting with clean data in the required form.
- The education and expertise of data scientists can (in some cases) provide less emphasis on experimental designs and theoretical inferences, focusing instead on systems, algorithms, and applications.

With all of that said, I will use the terms Statistics and Data Science interchangeably throughout the course.

Data Science Has Its Own Challenges



Many projects require very specific combinations of these skills. No one is a master of everything, but having a vision for your development can help immensely.

Of course, your personal plan may run into some practical challenges along the way.

Don’t Say I Didn’t Warn You

“80-90% of my time on a project is spent on cleaning up messy data sets. I would really like to spend more time doing analyses.”

– Origin: Every data scientist out there.

Yes, But Data Cleaning Is Important

- You can't generate good results without good data.
- You can't understand your data without understanding the mess it came from.
- Your detective work can improve many facets of the organization and reduce many of its risks.

Guardians of the Process

- Good decisions require good analyses.
- Good analyses require good data.
- Good data require a good process to ensure that the quality is robust.

Your role in an organization's work can be pivotal to its success. That all starts with understanding the data, which you will be uniquely qualified to do.

This All Sounds Very… Practical

- You may or may not have the data you want.
- The data may or may not have the kind of structure that leads to the analysis you want to perform.
- You may have to alter your plans many times along the way.

Principles of the Practical Practitioner

- Every new data set requires investigation.
- Every analysis must account for the structure and limitations of the data.
- Every project will depend on how you handle the twists and turns that you encounter along the way.

Getting the Answer can be the Easy Part

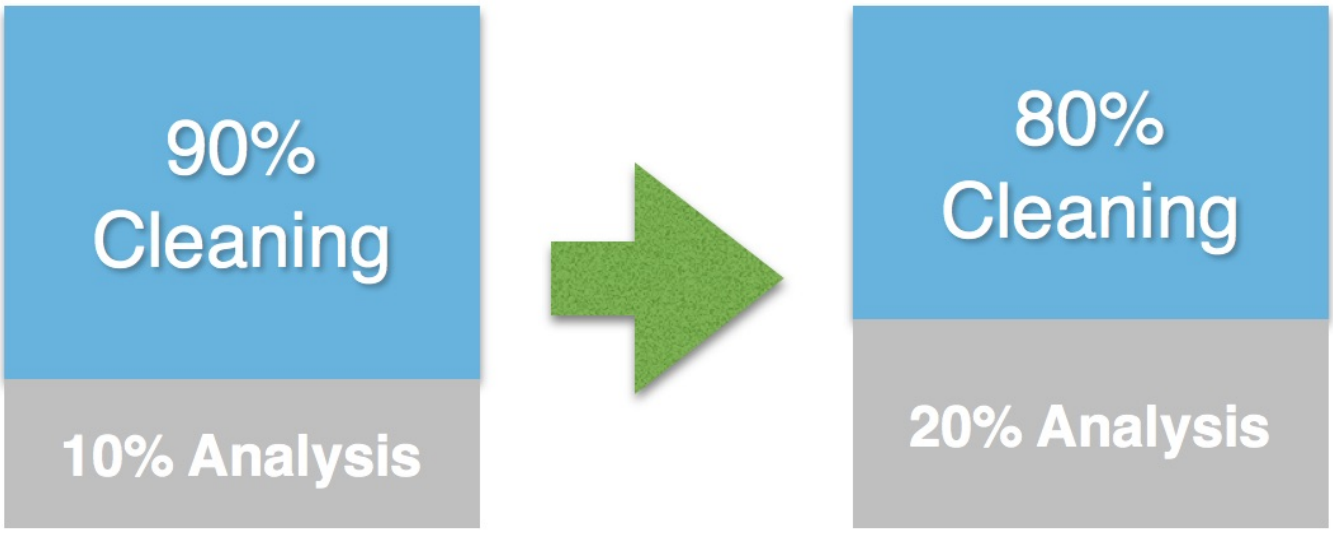
- Understanding the questions and challenges of the problem requires more effort than we initially think it should.
- Structuring the data you have into the form that you want is where most of the work lies.
- Then, after you do answer the question, communicating the results is what leads to a more comprehensive understanding of the problem – and to good decisions for your organization.

But Most Data Science Courses are Theoretical

- It is important to learn and understand the methods of analysis.
- You will succeed to the level that your skills – with computers, communication, knowledge about the field, etc. – will take you.
- However, every project's success ultimately hinges upon resolving the practical issues as much as the theoretical ones.

With that in mind, this course intends to provide an important complement to your theoretical and engineering curricula!

Better Yet, Opportunities Abound



- If you can improve your efficiency in data cleaning by just over 10%, you can **double your capacity** for analysis.
- If you recognize design issues early on, you can avoid a lot of fumbles and restarts later on.
- By honing your technical skills and gaining experience, **performance improvements of an order of magnitude** or more are within reach.

The Path to Success in Data Science

- Learning to investigate data in a manner that builds greater understanding and identifies challenges.
- Using computational tools that handle large, complex, and messy data sets in efficient ways.
- Developing coding skills that facilitate and simplify the work.
- Building processes that lead to enhanced accuracy, reliability, and accountability.
- Implementing practices that lead to greater productivity.

Sharing My Secrets

- Much of what we'll explore is rooted in my own career.
- That includes my successes and, perhaps more importantly, my mistakes along the way.
- We'll learn about challenges I faced and what I'd do differently with the benefit of hindsight.

What This Course Is All About

"When I was your age…"

- I had a limited sense of how analytical methods could be applied in broader settings.
- I was much less aware of the challenges of this work.
- More generally, we lacked the technical sophistication of today's tools.

I designed this course to provide more experience about the real challenges inherent in this work. I hope this course will provide you with some additional perspective about data science and help to prepare you for the opportunities that lie ahead.

Goals For The Semester

- Create a fast track for developing your expertise with R.
- Enhance your abilities to clean, organize, and understand data.
- Gain experience with the basic methods of data analysis.
- Develop some intuitions about statistical models.
- Learn how to interpret the results of analytical work.
- Learn how to *communicate* the results of analytical work.
- Explore the practice of data science across a variety of industries and applications.

How We'll Get There

- Intensive R training.
- Exploration of different challenges, case studies, and industries.
- Introduction of methods and techniques *in the context* of real projects.

This last point is crucial. Everyone learns better in context – by placing the material into real situations that you are likely to face. In this way, you can learn about theories, methods, programming tools, and industry applications in an integrated way.

How to be a Good Student

- Show up to class.
- **Practice, practice, practice**. Spend an hour a day on computer programming of some kind. Find your own data sets. Actively guide your own development.
- Start your homework early. Read it over an extra time before you turn it in.
- Talk to me and to the TAs (or send us e-mail). The more we learn about what you know and find challenging, the more we can do to help.
- Ask yourself questions. What can you do with this new technique? What makes sense, and what doesn't? How can you go about better understanding the problem you're trying to solve?
- Emphasize communication: How would you explain this to someone who knows the methods but not the domain? Who knows the domain but not the methods? Who is hearing about this for the first time? Learn to adjust your presentation to suit the audience.

Maybe You're Already a Good Student!

- Some of the content **may already be familiar** to you. That doesn't mean it has to be boring! Look at what you know with fresh eyes, and you will likely refine your understanding. Think more deeply about the applications or connecting the material you know to what you are trying to learn.

- Different students are starting off in different places. I will teach to everyone's needs as best as I can.

- The projects will provide opportunities for you to challenge yourself. If you feel like you know the material well, then show me what you can do!

Now it's time to dig in!

Why R?

- About as good as it gets for working with data.
- Highly interactive – you can easily explore your data while you are programming.
- A fully configurable programming language.
- Easy to use tools that minimize your labor.

Advantages of R

- Free and Open Source
- Reading Data
- Exploring Data
- Cleaning Data
- Analyses
- Graphics
- Building Models
- Building Tools
- Creating Reports

R is great for every aspect of data science!

What We'll Build

- Fast processing applications with the **data.table** package.
- Reproducible written reports using **rmarkdown**.
- Dynamic reporting engines using **shiny** and **flexdashboard**.

R Has Evolved and Is Evolving

Along the way, R has overcome many earlier struggles:

- Not Widely Used ———> Increasingly popular!
- Slow and inefficient ———> Highly competitive
- Lacks connections to Proprietary Systems ———> Connections abound.
- Poor GUI ———> A useful and highly configurable GUI
- Could not build usable tools ———> cutting edge applications.
- Poor Documentation ———> Well… R is getting better about that.

Evolution and Its Branches

- There are now many ways to program with R.
- Some older structures are not very useful – or may create complications
- The most modern structures are competing with each other.

A Downside to Open Source Languages

- R has numerous ways to do just about everything. The names of different packages can be very similar.
- The advantages and disadvantages of different approaches are not always obvious.
- No single package has a monopoly on all of the best features at any one time.
- The performance and popularity of a package can be weakly correlated.

Data Processing: Some Main Competitors

Most of what we'll focus on in the class starts with data processing. The main variants include:

- Classical data frames and processing methods
- **tibble** data frames (**tidyverse** package) with processing methods from **dplyr**.
- **data.table** extensions to data frames with their own associated processing methods.

My Point of View

Over time, everyone develops their own style of coding. My personal approach prioritizes productivity. To that end, I want to find methods that:

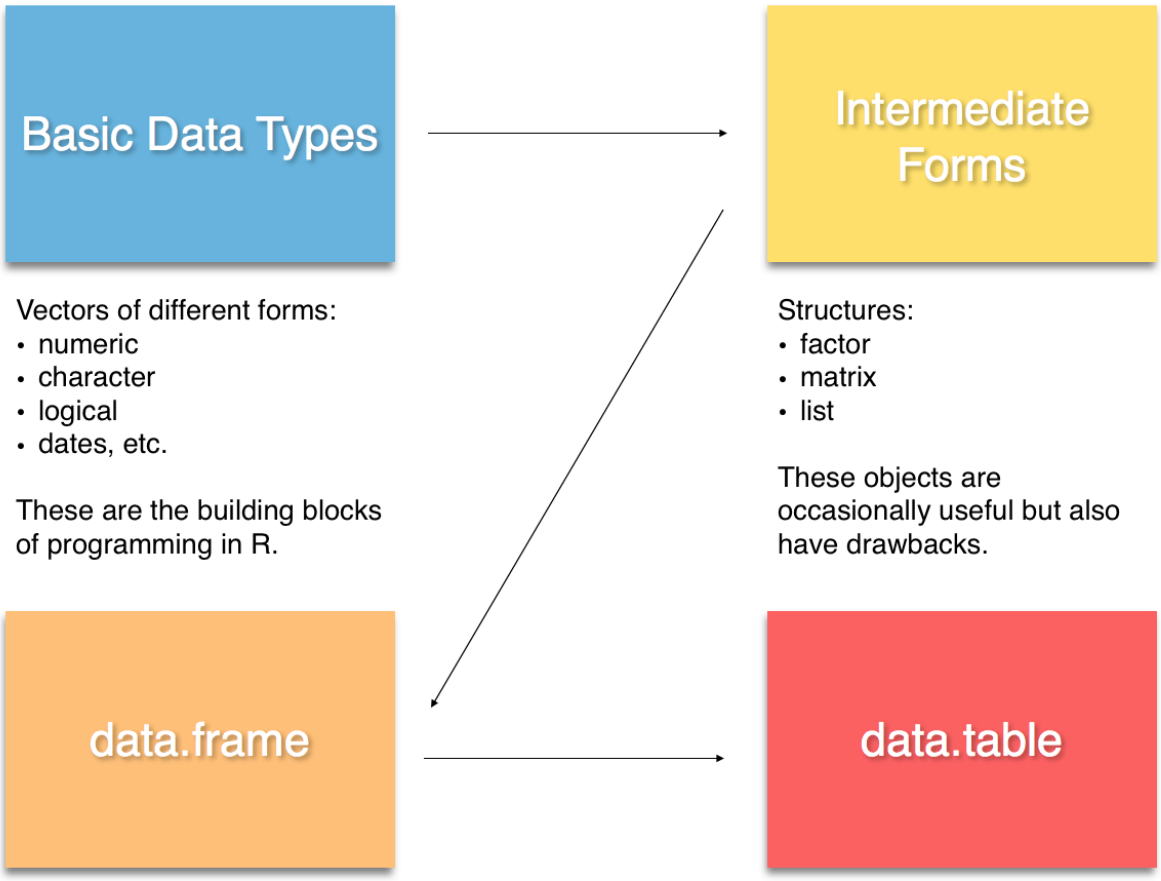
- Generate results as fast as possible.
- Handle large data sets well.
- Have a consistent syntax so that I can easily understand what to do (or remember what I did earlier).
- Enable *elegant* coding. The less I have to write, look up, and re-engineer, the better.

In my experience, the **data.table** package is best equipped to address these challenges and enable my work.

A Speed Test

https://github.com/Rdatatable/data.table/wiki/Benchmarks-:-Grouping

The Evolution of R's Data Types



Vectors of different forms:
- numeric
- character
- logical
- dates, etc.

These are the building blocks of programming in R.

Structures:
- factor
- matrix
- list

These objects are occasionally useful but also have drawbacks.

Classical data structure
- Rows for one record
- Columns for one variable
- Inputs to many methods (e.g. models).

A data.frame is a specialized list. Most of R's baseline capabilities were based on data.frame objects.

Powerful extension of data.frame object
- Fast and efficient
- Many advanced capabilities
- Concise syntax
- Elegant code

data.table enables the most powerful computations on large data files while maintaining a link to a data.frame's uses in R.

Evolution's Vestigial Forms

- The Intermediate data types, such as factors, matrices, and lists, and even data.frames, were once useful and the state of the art.
- However, all of these intermediate data types have drawbacks. They can be useful in limited contexts, but in many ways they have been surpassed.
- **data.table** objects have provided a path toward overcoming R's shaky early ground.

What about dplyr?

- A popular alternative to **data.table** is processing data with the **dplyr** package.

- Relative to **data.table**, dplyr is not as fast or capable of handling larger data sets. data.table's methods are often several times faster and can work more effectively on large amounts of data.

- However, the dplyr package does have its merits, e.g. working with database connections. Some people prefer the syntax as well.

Personally, I have found that the benefits of data.table's performance enhancements are compelling. Some of the homework assignments we'll work on by the end of the semester will start from large files. Even dplyr's designers have acknowledged the differences in performance on larger applications. And analyzing big data sets is increasingly a routine part of this work!

Meanwhile, I prefer data.table's unified approach to processing data with a relatively small number of powerful tools.

You can dplyr if you want to

- Other courses you take might teach from dplyr's methods.
- Ultimately, how you program is up to you.
- However, because of the clear differences in their capabilities, I will teach the course with a heavy emphasis on **data.table**'s methods. It's OK if you choose to do things differently, but it's worth evaluating these choices on their merits.

What About Python?

- Python is great, too. Relative to many other data processing and analytical languages, R and Python really stand out from the pack.
- You will certainly be learning Python in some of your other data science courses.
- The best thing you can do is fully explore the capabilities of these tools so that you can select the ones that are most appropriate for the challenge at hand.

Time for a Break

```
library(cowsay)
say(what = "We'll be right back!", by = "cow")
```

Developing A Good Style

- Make your code easy to read.
- Use comments to explain your work.
- Keep it organized.
- Make everything you write as *reusable* as possible.
- Be consistent!

Producing elegant and readable code is important. You want others to be able to easily understand your work. But guess who will read your code the most? **You will**, probably by a factor of 100. So the most important thing is creating a style that supports your own development with good habits.

Example Data: Asthma Patients

```
library(data.table)
dat <- fread(input = "simulated asthma data.csv")
print(dat)
```

```
             id     geog age inhaler controller spacer ER_1yr hosp_1yr
    1: DyZ8qxtW    Urban  21       0          1      0      0        0
    2: ofgjiI3n Suburban  12       1          1      0      0        0
    3: q5z67pHi Suburban  34       1          0      0      0        0
    4: qVNFTTdZ    Urban   2       1          0      0      3        0
    5: kyonZBzv    Urban   3       1          1      0      3        3
   ---
 9996: W7r8vxz0 Suburban  44       1          1      0      3        0
 9997: nLrxIj1M    Urban  22       1          1      1      1        2
 9998: C0PqDHrS    Urban   7       1          1      0      1        1
 9999: R9m7HMOA    Urban  17       0          1      1      0        0
10000: ZCmadlqQ    Urban  30       1          1      1      0        0
```

What are these Variables?

- **id**: a unique identifier for the patient.
- **age**: the patient's age at the beginning of a study.
- **geog**: What type of geographic region (Urban, Suburban, or Rural) the patient lives in.
- **inhaler**: this is a medicine (albuterol) that a patient can use to relieve asthma symptoms as they occur.
- **controller**: this is a steroid medicine that a patient takes daily as a means of reducing the onset of symptoms.
- **spacer**: this is a device that attaches to the inhalers and controllers. It contains the aerosolized medicine in a tube so that more of the medicine reaches the lungs when inhaled.
- **ER_1yr**: From the beginning of the study, how many trips to the Emergency Room did the patient have?
- **hosp_1yr**: From the beginning of the study, how many Hospital Admissions did resulted from trips to the Emergency Room? Hospital admissions typically result from more serious circumstances, with longer stays and higher costs.

data.table Lessons

Most **data.table** operations on an object **dat** can be reduced to some combination of three steps:

dat[i, j, by]

with the following actions:

- **i**: a statement of which *rows* of the data set to include.
- **j**: the calculations to perform on different *columns* of the data set. These calculations are only performed on the rows selected in the **i** step.
- **by**: how to aggregate the results.

We will discuss the details of all of these steps.

Selecting Rows Using i

```
dat[1, ]
```

```
         id  geog age inhaler controller spacer ER_1yr hosp_1yr
1: DyZ8qxtW Urban  21       0          1      0      0        0
```

```
dat[3:4]  ## Note that the first comma is optional but helpful.
```

```
         id     geog age inhaler controller spacer ER_1yr hosp_1yr
1: q5z67pHi Suburban  34       1          0      0      0        0
2: qVNFTTdZ    Urban   2       1          0      0      3        0
```

```
dat[c(8, 4931), ]
```

```
         id  geog age inhaler controller spacer ER_1yr hosp_1yr
1: KA2KTs9K Rural  25       1          1      0      0        0
2: 8XnBptcm Urban   0       1          1      1      0        0
```

This is an excellent way to show the record for 1 patient or a specific subset of them. Specificying a numeric vector in the **i** step will select the *row indices* to include.

More Advanced Subsetting in i

Any kind of logical operation is allowed. This will display all of the records for 2-year old patients:

```
dat[age == 2, ]
```

```
           id     geog age inhaler controller spacer ER_1yr hosp_1yr
  1: qVNFTTdZ    Urban   2       1          0      0      3        0
  2: NuhCSnZ4    Urban   2       0          1      0      0        0
  3: xzKqks7M    Urban   2       1          1      0      1        1
  4: SwWYasaG    Urban   2       0          1      0      0        1
  5: Vjzk9VeT Suburban   2       1          1      1      0        0
 ---
279: YhKP6LtT    Urban   2       1          0      1      0        0
280: jDopwOcO Suburban   2       1          1      0      0        2
281: iwtjNKJ0    Urban   2       1          1      0      1        1
282: Ng9CGJJE    Urban   2       0          0      0      0        0
283: 8Pk0ymqE Suburban   2       1          1      1      0        2
```

Subsetting With the %in% Operator

This will display all of the records for patients who are at least 60 years old and live in Suburban or Rural settings:

```
dat[age >= 60 & geog %in% c("Suburban", "Rural")]
```

```
           id     geog age inhaler controller spacer ER_1yr hosp_1yr
  1: CwlbQtR7 Suburban  61       0          0      0      1        1
  2: VT9TGuiF    Rural  61       1          0      0      2        0
  3: 78Nf8u9z Suburban  71       1          1      0      3        0
  4: NLWsQtmv    Rural  66       0          1      1      0        0
  5: vP19mjum    Rural  71       1          0      0      0        1
 ---
432: qlNRHwCp Suburban  73       0          1      1      0        0
433: uAEjVAoC Suburban  73       1          1      1      2        0
434: fIV9FRGC Suburban  75       1          0      1      0        0
435: 79kdyeJ7 Suburban  65       1          1      1      0        0
436: SyrG31Xz    Rural  85       1          1      1      0        0
```

More Subsetting

This will show the patients age 18 or younger who had at least one ER visit or hospital admission within the first year after diagnosis.

```
dat[age <= 18 & (ER_1yr >= 1 | hosp_1yr >= 1), ]
```

```
            id     geog age inhaler controller spacer ER_1yr hosp_1yr
   1: qVNFTTdZ    Urban   2       1          0      0      3        0
   2: kyonZBzv    Urban   3       1          1      0      3        3
   3: oGtqk7C5    Urban  16       1          1      0      1        2
   4: 3UvZXccZ    Urban   1       1          0      1      1        0
   5: z0Bb2tJ4 Suburban   0       0          1      0      1        0
  ---
2513: fOo91bB2    Rural  10       1          1      0      0        1
2514: 3rZNO2uB    Urban   6       1          1      0      1        1
2515: svAJPixR Suburban   3       1          1      1      1        0
2516: BfSnViQt Suburban   0       1          1      1      2        0
2517: C0PqDHrS    Urban   7       1          1      0      1        1
```

Column Operations in j

The j step allows you:

- to display the values of one or more variables;
- to perform new calculations;
- to add to or remove variables from the data.table object.

All of these calculations are performed **on the rows selected in the i step**.

Selecting a Variable

This will display the ages of the first 5 patients:

```
dat[1:5, age]
```

```
[1] 21 12 34  2  3
```

```
dat[1:5, .(age)]
```

```
   age
1:  21
2:  12
3:  34
4:   2
5:   3
```

What is the difference between these two operations?

Selecting with and without .()

- dat[, age] returns a vector
- dat[, .(age)] returns a **data.table** with 1 column (age).
- Multiple variables can be selected in a single operation – but only inside of .() in the **j** step.

Selecting Two or More Variables with .()

Show the age and geography of the first five patients:

```
dat[1:5, .(age, geog)]
```

```
   age     geog
1:  21    Urban
2:  12 Suburban
3:  34 Suburban
4:   2    Urban
5:   3    Urban
```

Calculations in j on Selected Variables

You can also do computations within j:

```
dat[, .(median(x = age, na.rm = TRUE), mean(x = ER_1yr,
    na.rm = TRUE))]
```

```
   V1     V2
1: 22 0.7093
```

Adding Column Names

Notice that these results may not have column names. But you can specify them yourself:

```
dat[, .(median_age = median(x = age), mean_ER_1yr = mean(x = ER_1yr))]
```

```
   median_age mean_ER_1yr
1:         22      0.7093
```

i and j, Together

```
dat[geog == "Urban", .(median_age_urban = median(x = age,
    na.rm = TRUE), mean_ER_1yr_urban = mean(x = ER_1yr,
    na.rm = TRUE))]
```

```
   median_age_urban mean_ER_1yr_urban
1:               18         0.8349169
```

This calculation combines the **i** and **j** steps. It says:

- Filter the data.table to cases of Urban geography.
- Then compute the median age and the mean number of ER visits (while removing missing values).
- Display the results as a new data.table object.

Matching Names to Subsets

What happens if we leave the i step blank in the previous code?

```
dat[, .(median_age_urban = median(x = age, na.rm = TRUE),
    mean_ER_1yr_urban = mean(x = ER_1yr, na.rm = TRUE))]
```

```
   median_age_urban mean_ER_1yr_urban
1:               22            0.7093
```

Now we have a computation over **the whole original data.table dat**, including all geographies. The names of the new columns **still have the urban label**. Be careful to make sure that your names match up with your computations!

Supplying Your Own Functions

A patient is considered controlled if they have stayed out of the hospital for a year. The **mean_controlled** function computes the percentage of patients who met this target. Show the percentage in an Urban setting who were controlled at 1 year.

```
mean_controlled <- function(x) {
    controlled <- 1 * (x == 0)
    return(mean(x = controlled, na.rm = TRUE))
}
dat[geog == "Urban", .(urban_controlled = mean_controlled(x = hosp_1yr +
    ER_1yr))]
```

```
   urban_controlled
1:        0.3939034
```

Creating a Variable in the data.table

```
dat[, takes_meds := 1*(inhaler == 1 | controller == 1)]
```

Under the hood, this method creates the new column *by reference*. This is much faster and memory efficient than the classic approach of data.frame objects, which would read as

```
dat$takes_meds = 1 * (dat$inhaler == 1 | dat$controller ==
    1)
```

Adding a column **by reference** only allocates new memory for the new column. This is much faster because it only does the necessary work to store the values. The classical data.frame method involves some degree of additional copying of values from one place to another, which can increase the running time by a factor of 15-20 in large data sets.

Two Forms of Variable Assignments

Equivalent Forms

Within a **data.table** object, there are two ways to assign a new variable. In this example, we are creating a column called **takes_meds** to classify patients based on whether they take at least one of the medicines (inhaler or controller) or not.

Form 1: I always write it like this!

```
dat[, takes_meds := 1 * (inhaler == 1 | controller == 1)]
```

Form 2: I never write it like this!

```
dat[, `:=`(takes_meds, 1 * (inhaler == 1 | controller == 1))]
```

Showing the Names of the data.table

These two different formats are equivalent. We can see below that takes_meds was added as a new column of dat:

```
names(dat)
```

```
[1] "id"          "geog"        "age"         "inhaler"     "controller"
[6] "spacer"      "ER_1yr"      "hosp_1yr"    "takes_meds"
```

Results of Adding a Variable

```
dat[, .(inhaler, controller, takes_meds)]
```

```
       inhaler controller takes_meds
    1:       0          1          1
    2:       1          1          1
    3:       1          0          1
    4:       1          0          1
    5:       1          1          1
   ---
 9996:       1          1          1
 9997:       1          1          1
 9998:       1          1          1
 9999:       0          1          1
10000:       1          1          1
```

It is always worthwhile to inspect your designs. Check some cases to make sure that the computation was working effectively.

Removing a Variable with := NULL in j

```
dat[, takes_meds := NULL]
dat[1:5,]
```

	id	geog	age	inhaler	controller	spacer	ER_1yr	hosp_1yr
1:	DyZ8qxtW	Urban	21	0	1	0	0	0
2:	ofgjiI3n	Suburban	12	1	1	0	0	0
3:	q5z67pHi	Suburban	34	1	0	0	0	0
4:	qVNFTTdZ	Urban	2	1	0	0	3	0
5:	kyonZBzv	Urban	3	1	1	0	3	3

Counting Rows with .N in j

How many rows are there in the data set?

```
dat[, .N]
```

```
[1] 10000
```

How many rows are there for Urban patients?

```
dat[geog == "Urban", .(num_Urban = .N)]
```

```
   num_Urban
1:      5052
```

Challenges of the j Step

So far, we have the tools for many computations in the j step. However, this approach has a few shortcomings:

- You must know all of the variable names;
- Enumeration of each variable is required;
- It's not immediately dynamic enough to handle changing structures.

Fortunately, all of these obstacles can be overcome with some additional tools.

Dynamic j with eval and get

- **eval(expr)**: returns the *name* of a variable
- **get(x)**: returns the *values* of a variable.

This turns out to have many useful applications working with data.table objects.

Example: Add a New Variable in a Function with eval

```
name.and.year <- function(dat, variable.name, year, value){
  require(data.table)
  setDT(dat)

  the.new.name <- sprintf("%s_%d", variable.name, year)
  dat[, eval(the.new.name) := value]
  return(dat[])
}
```

```
dat <- name.and.year(dat = dat, variable.name = "is_Urban",
    year = 2019, value = (dat[, geog == "Urban"]))
print(dat[1:5, .(geog, is_Urban_2019)])
```

```
       geog is_Urban_2019
1:    Urban          TRUE
2: Suburban         FALSE
3: Suburban         FALSE
4:    Urban          TRUE
5:    Urban          TRUE
```

In other words, you can add a new variable with a dynamic name (that might change from year to year or be supplied by a user).

Example: Calculations in a Function with get

```
mean.and.N <- function(dat, variable.name) {
    require(data.table)
    setDT(dat)
    tab <- dat[, .(mean_value = mean(get(variable.name),
        na.rm = TRUE), N = .N)]
    return(tab[])
}
mean.and.N(dat = dat[geog == "Urban"], variable.name = "inhaler")
```

```
   mean_value    N
1:  0.6256928 5052
```

How I Write My Programs

```
age.name <- "age"
age.group.name <- "age.group"
age.group.cuts <- c(0, 5, 10, 18, 30, 50, 65, 120)
spacer.name <- "spacer"
library(Hmisc)
```

```
dat[, eval(age.group.name) := cut2(x = get(age.name), cuts = age.group.cuts)]
sprintf("The mean age was %.2f.", dat[, mean(get(age.name), na.rm = TRUE)])
```

```
[1] "The mean age was 24.33."
```

```
dat[, .(mean_spacer = mean(get(spacer.name))), keyby = age.group.name]
```

```
   age.group mean_spacer
1: [  0,  5)   0.3501006
2: [  5, 10)   0.3006924
3: [ 10, 18)   0.3091977
4: [ 18, 30)   0.3102616
5: [ 30, 50)   0.3367468
6: [ 50, 65)   0.3765823
7: [ 65,120]   0.4336570
```

Why Do This Extra Work?

Writing my programs like this, I can change all of the downstream steps by simply changing the constants at the top. If the names of the variables or the cut-offs change, I don't have to make the changes in dozens of places.

We will revisit this topic in a few lectures to examine the reproducibility of the code.

Supercharging the j Step with lapply and .SDcols

```
clinical.variables <- c("age", "inhaler", "controller",
    "spacer")
dat[, lapply(X = .SD, FUN = "mean", na.rm = TRUE), .SDcols = clinical.variables]
```

```
     age inhaler controller spacer
1: 24.33   0.668     0.6447 0.3298
```

The .SDcols argument allows you to name (or index) multiple columns of the data set.

Then, the **lapply** function in the j step applies the specificied function (mean) to each of the .SDcols columns. You can also specificy additional arguments to the mean function like na.rm = TRUE.

.SDcols is Especially Fast and Efficient

- It is much better to use this than to run a for loop
- It also returns all of the results in one place – no aggregation is necessary!
- Better yet, you don't have to repeat the steps of the calculation across a loop or multiple enumerations. One call of lapply can run a function on up to every column of the data.table.

Programmatically Selecting Variables for .SDcols

```
outcome.variables.index <- grep(pattern = "_1yr", x = names(dat))
print(x = outcome.variables.index)
```

```
[1] 7 8
```

```
dat[, lapply(X = .SD, FUN = "mean", na.rm = TRUE), .SDcols = outcome.variables.index]
```

```
   ER_1yr hosp_1yr
1: 0.7093   0.4166
```

This feature is especially nice. You don't have to list out all of the names of the variables directly. Instead, you can search for patterns. Keep that in mind as you start to work with larger files that may have thousands of column names!

Numeric or Character Values of .SDcols

The .SDcols can either be

- Numeric vectors of the index columns, or
- Character vectors specifying the names of the columns.

Let's revise the previous calculation to incorporate a character vector into .SDcols:

```
outcome.variables.character <- names(dat)[outcome.variables.index]
print(x = outcome.variables.character)
```

```
[1] "ER_1yr"   "hosp_1yr"
```

```
dat[, lapply(X = .SD, FUN = "mean", na.rm = TRUE), .SDcols = outcome.variables.character]
```

```
   ER_1yr hosp_1yr
1: 0.7093   0.4166
```

Eliminating Factors In One Fell Swoop

```
convert.factor.to.character <- function(x) {
    if (is.factor(x)) {
        return(as.character(x))
    } else {
        return(x)
    }
}
dat <- dat[, lapply(X = .SD, FUN = "convert.factor.to.character")]
```

If you'd prefer not to use factor variables, then this is a simple way to make the conversion.

Computing Rates of Missing Data

```
mean_missing <- function(x) {
    return(mean(x = is.na(x)))
}
dat[, lapply(X = .SD, FUN = "mean_missing")]
```

```
   id geog age inhaler controller spacer ER_1yr hosp_1yr is_Urban_2019
1:  0    0   0       0          0      0      0        0              0
   age.group
1:         0
```

```
dat[, lapply(X = .SD, FUN = "mean_missing"), .SDcols = 1:ncol(dat)]
```

```
   id geog age inhaler controller spacer ER_1yr hosp_1yr is_Urban_2019
1:  0    0   0       0          0      0      0        0              0
   age.group
1:         0
```

When .SDcols is not included, its default includes all of the columns of the data.table.

A rounding function

```
round.numerics <- function(x, digits) {
    if (is.numeric(x)) {
        x <- round(x = x, digits = digits)
    }
    return(x)
}
```

This function is applied to a single vector of data. It will:

- Check the structure of the vector.
- If the vector is numeric, it will be rounded to the specified number of digits.
- Otherwise, it will be returned untouched.

The effect here is interesting – you can apply **round.numerics** to every column of data to round the numeric vectors without impacting the other columns. This is an excellent way to display tables in reports.

Easily Rounding Your Results

For rural patients under 5 years old, show the mean 1-year outcomes, all rounded to 3 decimal places.

```
outcome.variables <- grep(pattern = "_1yr", x = names(dat))
rural.under5.results <- dat[geog == "Rural" & age < 5, lapply(X = .SD,
    FUN = "mean", na.rm = TRUE), .SDcols = outcome.variables]
rural.under5.results[, lapply(X = .SD, FUN = "round.numerics",
    digits = 3)]
```

```
   ER_1yr hosp_1yr
1:  0.768    0.317
```

The by Step in data.table Objects

Let's revisit our example of subsetting to patients in urban settings:

```
dat[geog == "Urban", .(median_age_urban = median(x = age,
    na.rm = TRUE), mean_hosp_1yr_urban = mean(x = hosp_1yr,
    na.rm = TRUE))]
```

```
   median_age_urban mean_hosp_1yr_urban
1:               18           0.5229612
```

Subsetting To All Groups Is Laborious

- Subsetting to urban patients in i;
- Providing urban labels in j;
- Calculating all of the column results in j;
- Repeating these steps for the other subgroups (suburban and rural).
- Aggregating the results into one table.

Grouping Made Easy with the by Step

```
values.by.geog <- dat[, .(median_age = median(x = age, na.rm = TRUE),
    mean_hosp_1yr = mean(x = hosp_1yr, na.rm = TRUE)), by = "geog"]
values.by.geog[, lapply(X = .SD, FUN = "round.numerics",
    digits = 3)]
```

```
       geog median_age mean_hosp_1yr
1:    Urban         18         0.523
2: Suburban         25         0.308
3:    Rural         32         0.309
```

1-Year Outcomes By Geography

For patients under the age of 18, what are the mean outcomes at 1 year, split by the geographic density, rounded to 3 decimal places??

```
outcomes.by.geog <- dat[age < 18, lapply(X = .SD, FUN = "mean",
    na.rm = TRUE), .SDcols = outcome.variables, by = "geog"]
outcomes.by.geog[, lapply(X = .SD, FUN = "round.numerics",
    digits = 3)]
```

```
       geog ER_1yr hosp_1yr
1: Suburban  0.654    0.355
2:    Urban  0.962    0.600
3:    Rural  0.674    0.326
```

Knowledge Check

What is the difference between the **i** step and the **by** step?

Answer

- The i step filters what *comes in* to the calculation from the existing data.table object.
- The by step organizes what *goes out* of the calculation into a new data.table object.
- The results of the j and by steps **depend on** what is allowed in by the i step.

Multiple Variables in the by Step.

Patients are considered pediatric if they're under 18 and otherwise not. For each combination of geography and pediatric status, show the mean rates of use for spacers, inhalers, and controllers, all rounded to 3 decimal places.

```
dat[, ped := 1*(age < 18)]
```

```
rate.variables <- c("spacer", "inhaler", "controller")
by.variables <- c("ped", "geog")
usage.tab.ped.geog <- dat[, lapply(X = .SD, FUN = "mean",
    na.rm = TRUE), .SDcols = rate.variables, by = by.variables]
res <- usage.tab.ped.geog[, lapply(X = .SD, FUN = "round.numerics",
    digits = 3)]
res
```

```
   ped     geog spacer inhaler controller
1:   0    Urban  0.235   0.626      0.573
2:   1 Suburban  0.474   0.788      0.749
3:   0 Suburban  0.468   0.802      0.743
4:   1    Urban  0.234   0.626      0.566
5:   0    Rural  0.345   0.585      0.680
6:   1    Rural  0.412   0.555      0.696
```

Sorting a data.table with setorderv

The previous table is nice, but it could be reordered in a more logical way. The **setorderv** command lets you sort a data.table by multiple variables (in order):

```
setorderv(x = res, cols = by.variables, order = c(-1, 1))
res
```

```
   ped     geog spacer inhaler controller
1:   1    Rural  0.412   0.555      0.696
2:   1 Suburban  0.474   0.788      0.749
3:   1    Urban  0.234   0.626      0.566
4:   0    Rural  0.345   0.585      0.680
5:   0 Suburban  0.468   0.802      0.743
6:   0    Urban  0.235   0.626      0.573
```

Using the by step will display the results in the order that they appear in the table. The **setorderv** function allows you to sort a data.table by the columns you specificy.

The **order** parameter allows you to sort in ascending (1) or descending (-1) order for each variable.

Or Use keyby

```
usage.tab.ped.geog <- dat[, lapply(X = .SD, FUN = "mean",
    na.rm = TRUE), .SDcols = rate.variables, keyby = by.variables]
res <- usage.tab.ped.geog[, lapply(X = .SD, FUN = "round.numerics",
    digits = 3)]
res
```

```
   ped     geog spacer inhaler controller
1:   0    Rural  0.345   0.585      0.680
2:   0 Suburban  0.468   0.802      0.743
3:   0    Urban  0.235   0.626      0.573
4:   1    Rural  0.412   0.555      0.696
5:   1 Suburban  0.474   0.788      0.749
6:   1    Urban  0.234   0.626      0.566
```

- keyby is a faster operation and requires one less line of code.
- However, keyby can only sort in ascending order.

Inverting the .SDcols table

Rather than putting all of the calculations in separate columns, sometimes it can be useful to create an overall column of summarized values **across the different measures**. Using .SDcols in a slightly different way can facilitate that:

```
rates.ped.geog <- dat[, .(variable = rate.variables, mean_usage = as.numeric(lapply(X = .SD,
    FUN = "mean", na.rm = TRUE))), .SDcols = rate.variables,
    keyby = c("geog")]

rates.ped.geog[, lapply(X = .SD, FUN = "round.numerics",
    digits = 3)]
```

```
       geog   variable mean_usage
1:    Rural     spacer      0.360
2:    Rural    inhaler      0.578
3:    Rural controller      0.684
4: Suburban     spacer      0.470
5: Suburban    inhaler      0.797
6: Suburban controller      0.745
7:    Urban     spacer      0.235
8:    Urban    inhaler      0.626
9:    Urban controller      0.570
```

The **as.numeric** calculation will convert lapply's **list** output into a numeric vector.

Advantages of the Design

Across all of the variables and geographies, which ones have mean usage rates under 0.4?

```
rates.ped.geog[mean_usage < 0.4, lapply(X = .SD, FUN = "round.numerics",
    digits = 3)]
```

```
    geog variable mean_usage
1: Rural   spacer      0.360
2: Urban   spacer      0.235
```

And That's the Power of R

- Fast, efficient results
- Elegant code
- Flexible structures

Now It's Time to Practice

- We will be using R throughout the semester.
- The data.table package has a learning curve. We will continue working on it.
- A copy of the simulated asthma data used is available on the course's website. Read it into R with the data.table function **fread**.
- There are great resources available to study. Start training up!