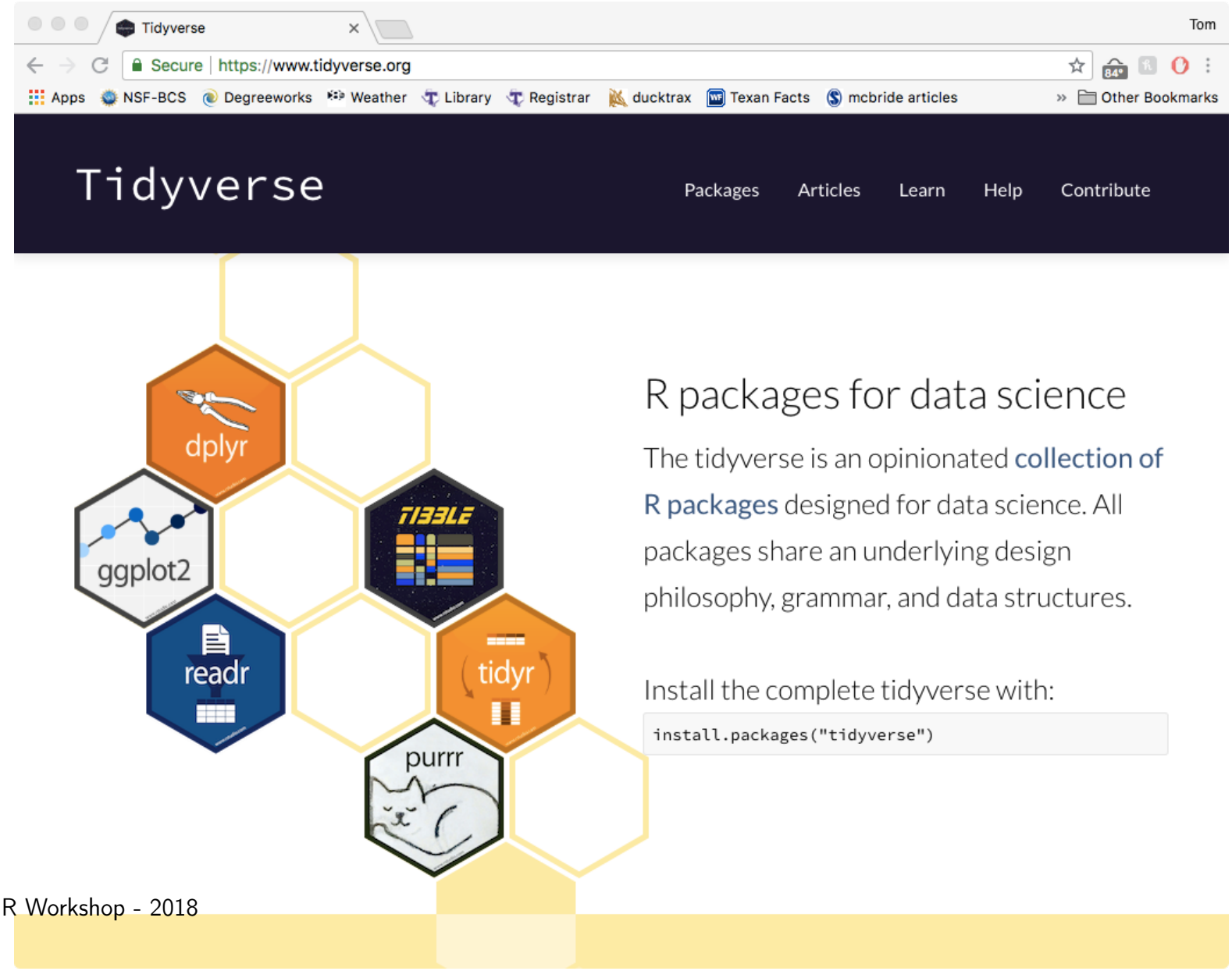
Session 3: The Tidyverse

Thomas J. Faulkenberry, Ph.D.

Tarleton State University


Tidyverse
Tom
Secure | https://www.tidyverse.org
Apps
NSF-BCS
Degreeworks
Weather
Library
Registrar
ducktrax
Texan Facts
mcbride articles
Other Bookmarks
Tidyverse
Packages
Articles
Learn
Help
Contribute
dplyr
ggplot2
tibble
readr
tidyr
purrr
R packages for data science
The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.
Install the complete tidyverse with:
install.packages("tidyverse")

Tidyverse packages

- `dplyr` provides easy tools for the most common data manipulation tasks. It is built to work directly with data frames, with many common tasks optimized by being written in a compiled language (C++).

- `tidyr` addresses the common problem of wanting to reshape your data for plotting and use by different R functions. Sometimes we want data sets where we have one row per measurement. Sometimes we want a data frame where each measurement type has its own column, and rows are instead more aggregated groups. Moving back and forth between these formats is nontrivial, and `tidyr` gives you tools for this and more sophisticated data manipulation.

- `ggplot2` is a powerful system for producing graphics efficiently

Getting started

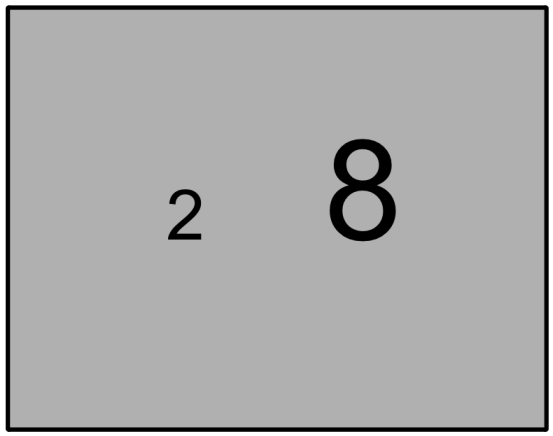
To begin, let's load a big data set and see the power of the Tidyverse in action:

```
library(tidyverse)
rawdata = read_csv("https://git.io/fNbny")
```

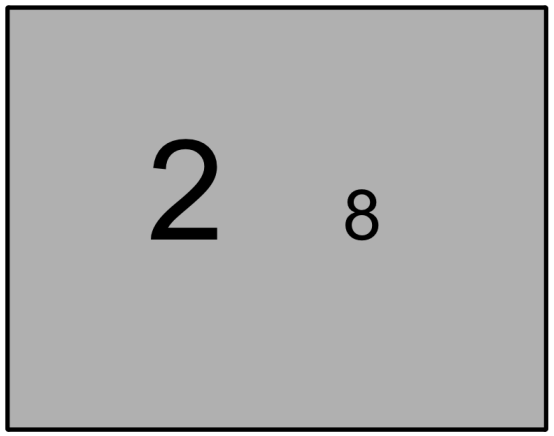
The data

This data set comes from Experiment 2 of Faulkenberry, Cruise, Lavro, and Shaki (2016). We presented people with pairs of single-digit numbers written in two different font sizes. We asked them to use a computer mouse to click on the *physically larger* of the pair in two experimental conditions (see below). We measured RT in milliseconds.

Congruent



Incongruent



Selecting columns and filtering rows

First, we're going to learn some of the most common `dplyr` functions (also called **verbs**): `select()`, `filter()`, `mutate()`, `group_by()`, and `summarize()`.

To choose *columns* of a data frame, use `select()`. The first argument to this function is the data frame (`rawdata`), and the subsequent arguments are the columns to keep.

```
select(rawdata, condition, error, RT)
```

Selecting columns and filtering rows

To choose *rows* based on specific criteria, use `filter()`

```
filter(rawdata, error==0)
```

Pipes

But what if you wanted to select and filter *at the same time*? There are many ways to do this, but the quickest and easiest is to use **pipes**.

Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset.

```
data = rawdata %>%
  filter(error==0) %>%
  select(subject, condition, RT)
```

Mutate

Frequently you will want to create *new columns* based on the values in existing columns. For example, we might want to express response time (RT) in seconds rather than milliseconds. For this, we'll use `mutate()`:

```
data %>%
  mutate(RT_sec = RT/1000)
```

Mutate

Another good use of `mutate()` is for *recoding* a variable. For example, the column `distance` currently has 4 values: 1, 2, 3, and 4. Suppose we want to recode this variable to have two values: close (distance=1 or 2) and far (distance=3 or 4). We can use `mutate()` along with `ifelse()` to do this:

```
data = data %>%
  mutate(dist = ifelse(distance==1 | distance==2, "close", "far"))
```

Summarize

Many data analysis tasks can be approached using the *split-apply-combine* paradigm: split the data into groups, apply some analysis to each group, then combine the results. `dplyr` makes this very easy using the functions `group_by()` and `summarize()`.

The following code chunk illustrates this:

```
data %>%
  group_by(condition, dist) %>%
  summarize(meanRT = mean(RT))
```

Summarize

Also, we can compute multiple statistics:

```
data %>%
  group_by(condition, dist) %>%
  summarize(meanRT=mean(RT), sd=sd(RT))
```

Advanced plotting using `ggplot2`

`ggplot` graphics are built *step by step* by adding new elements. Adding layers in this fashion allows for extensive flexibility and customization of plots.

To build a `ggplot`, we need to:

- pipe our data to the `ggplot()` function
- define *aesthetics* (`aes`) by selecting the variables to be plotted and the variables to define the presentation.
- add `geoms`, which are graphical representations in the plot (points, lines,bars).

Advanced plotting using ggplot2

Example 1: a boxplot

```
data %>%
  ggplot(aes(x=condition, y=RT)) +
  geom_boxplot()
```

Advanced plotting using `ggplot2`

Example 1.5: a *horizontal* boxplot

```
data %>%
  ggplot(aes(x=condition, y=RT)) +
  geom_boxplot() +
  coord_flip()
```

Advanced plotting using `ggplot2`

Example 2: a *histogram*

```
data %>%
  ggplot(aes(x=RT, group=condition)) +
  geom_histogram(aes(fill=condition))
```

Advanced plotting using ggplot2

Example 3: overlaid density plots

```
data %>%
  ggplot(aes(x=RT, group=condition)) +
  geom_density(aes(fill=condition))
```

Plotting summaries

Often, we are interested in the differences between condition means. We will demonstrate this using two types of plots: a bar plot, and a line plot.

First, lets look at a bar plot that demonstrates the difference in condition means between incongruent and congruent trials. Notice how we are using the "split-apply-combine" paradigm along with `ggplot` here:

```
data %>%
  group_by(condition) %>%
  summarize(meanRT=mean(RT)) %>%
  ggplot(aes(x=condition,y=meanRT)) +
  geom_bar(stat="identity", width=0.5)
```

Plotting summaries

Similarly, a small change can produce a line plot instead:

```
data %>%
  group_by(condition) %>%
  summarize(meanRT=mean(RT)) %>%
  ggplot(aes(x=condition, y=meanRT, group=1)) +
  geom_line() +
  geom_point()
```

Plotting summaries

What if we were interested in the differences in mean RT by distance? We can easily edit our code above to get a line plots for distance instead of condition:

```
data %>%
  group_by(dist) %>%
  summarize(meanRT=mean(RT)) %>%
  ggplot(aes(x=dist, y=meanRT, group=1)) +
  geom_line() +
  geom_point() +
  ylim(0,1500)
```

Plotting summaries

Both plots indicate that numerical distance doesn't seem to have much effect on RTs.

However, a more interesting plot might reveal something different! Lets see what happens when we plot BOTH condition and distance on the same plot:

```
data %>%
  group_by(condition, dist) %>%
  summarize(meanRT=mean(RT)) %>%
  ggplot(aes(x=dist, y=meanRT, group=condition)) +
  geom_line(aes(linetype=condition)) +
  geom_point(aes(shape=condition)) +
  ylim(1000,1500)
```

Plot themes

One of the fun things with ggplot is that you can EASILY change the theme of your plot with one or two lines of code:

```
data %>%
  group_by(condition, dist) %>%
  summarize(meanRT=mean(RT)) %>%
  ggplot(aes(x=dist, y=meanRT, group=condition)) +
  geom_line(aes(linetype=condition)) +
  geom_point(aes(shape=condition), size=2) +
  ylim(1000,1500) +
  theme_classic(16)
```