Tidying Data with R

The goal of this lecture is to delve deeper into common forms of data work. The reason we work with data is to find answers of interest to us, but the data we find or receive is rarely in the form we need. Oftentimes this impedes our ability to even begin such an analysis. Common issues with messy data and our topics for today:

1. Vectors/variables are of the wrong type

- Factors
- Dates

2. Strings have extra, unneeded information

- Extracting parts of a character string
- Fixing typos

3. Information in a table is difficult to extract

- Wide and long form data
- Reshaping data

Introduction to data and research goal

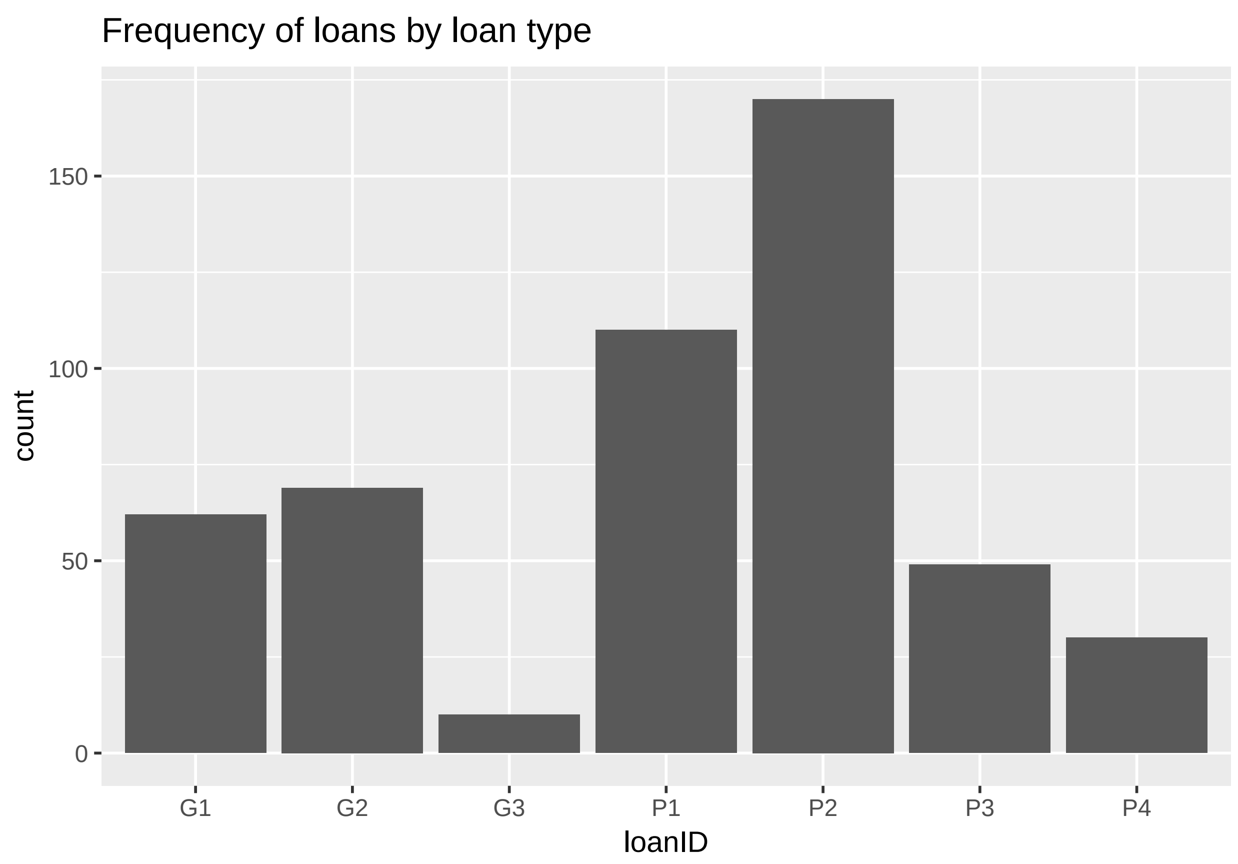
The research goal for this lecture is to study the effects of banks' characteristics on lending behavior. Some key measurements relevant to this research might be:
* Number of loans issued every month
* Type of banks that are issuing loans
* Proportion of loans, between private and government loans, being issued
* etc. Can you name some more?

To address our question(s) we will use what we have available to us, the banks dataset.

```r
head(banks)
```

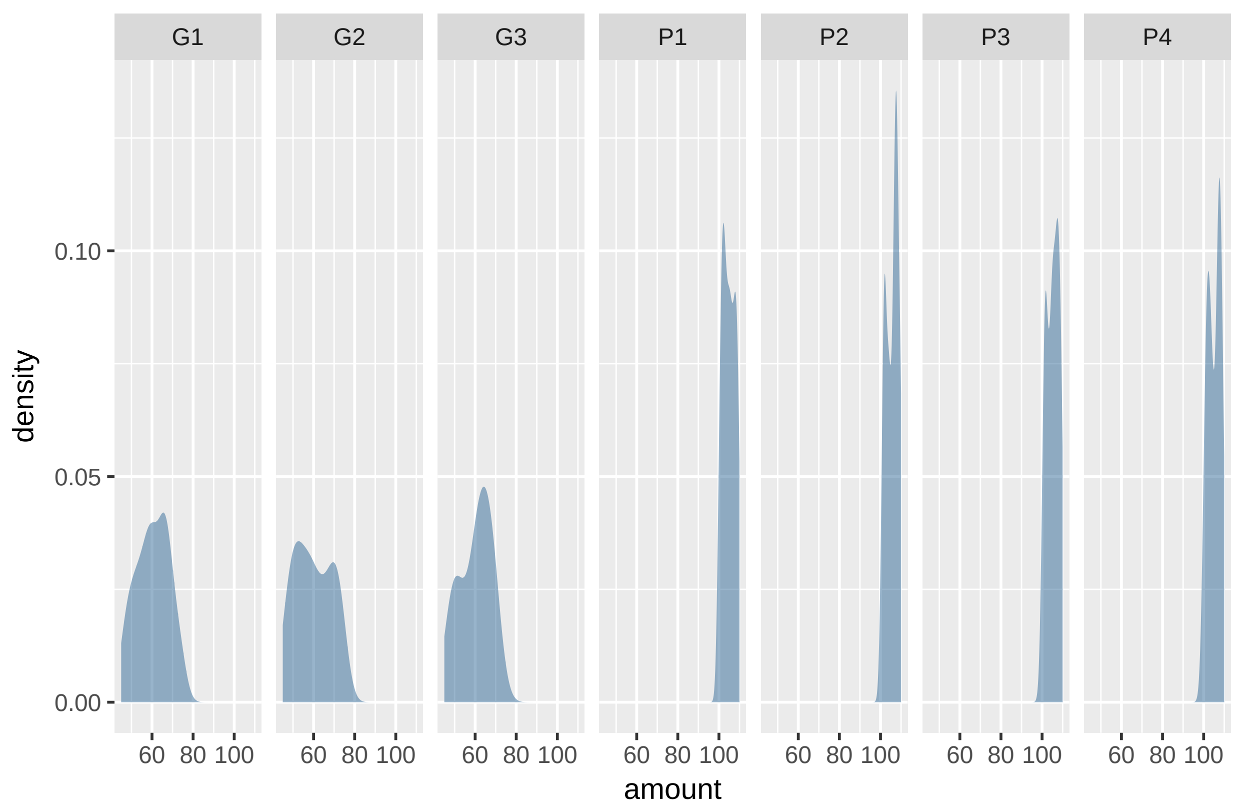
```
## # A tibble: 6 x 5
##         date bankID         type loanID amount
##        <chr>  <dbl>        <chr>  <chr>  <int>
## 1 2015-07-06   9714   Commercial     P1    105
## 2 2015-07-07   3351   Commercial     P2    102
## 3 2015-07-08   8459 Credit Union     P4    107
## 4 2015-07-09   4669   Commercial     P4    109
## 5 2015-07-10   2395   Commercial     P1    107
## 6 2015-07-11   3351   Commercial     G2     54
```

```r
ggplot(banks, aes(x = loanID)) +
  geom_bar(stat = "count") +
  labs(title = "Frequency of loans by loan type")
```



```r
ggplot(banks, aes(x = amount, fill = 1)) +
  geom_density(alpha = 0.5, color = NA) +
  facet_grid(. ~ as.factor(loanID)) +
  guides(fill = FALSE) +
  labs(title = "Distribution of daily loan amounts by type of loan")
```

Distribution of daily loan amounts by type of loan



Important questions to consider:

- What in banks seems useful?
- What seems unneeded?
- Can you think of any additional variables that would be useful for us?

Let's zoom in on the date variable. We know at least one of our measures of interest relies on knowing which time period each data point belongs to. The date variable that we have sort of accomplishes this, but it is not in an easy-to-use format.

1. Vectors/variables are of the wrong type

Dates in R

Dates may not initially seem like complicated objects, but they are defined by a fairly complex mix of geographical, political, astronomical, and mathematical factors. R treats dates as a uniquely-formatted vector type, represented as: "YYYY-MM-DD". The as.Date() function can convert strings into this date vector type. Since dates are a vector type, they can be assigned to objects like any other variable, and R can do most operations (that make sense) on dates, simply as part of the basic R package.

```
dates <- c(as.Date("2010-01-01"), as.Date("2010-12-31"))

print(dates)
```

```
## [1] "2010-01-01" "2010-12-31"
```

```r
mean(dates)
```

```
## [1] "2010-07-02"
```

```r
min(dates)
```

```
## [1] "2010-01-01"
```

```r
as.Date("2010-02-10") + 10
```

```
## [1] "2010-02-20"
```

R can convert many, many different looking strings into a date class using the "format" argument in as.Date:

```r
as.Date("1980-02-10")
```

```
## [1] "1980-02-10"
```

```r
as.Date("19800210", format = "%Y%m%d")
```

```
## [1] "1980-02-10"
```

```r
as.Date("10Feb80", format = "%d%b%y")
```

```
## [1] "1980-02-10"
```

```r
as.Date("9/18/2016", format = "%m/%d/%Y")
```

```
## [1] "2016-09-18"
```

Symbol	Meaning
%d	Day as a number
%a	Abbreviated Weekday
%A	Unabbreviated Weekday
%m	Month as a number
%b	Abbreviated month
%B	Unabbreviated month
%y	Two-digit year
%Y	Four-digit year

In-class exercise:

Convert the following character vector to a date vector.

```r
date_vec <- c("Day 5 of March, 2017", "Day 23 of May, 2017")
```

Format is both an argument and a function itself:

```r
Sys.Date()
```

```
## [1] "2017-07-28"
```

```r
format(Sys.Date(), "%d %B %Y")
```

```
## [1] "28 July 2017"
```

```r
format(Sys.Date(), "%d-%b")
```

```
## [1] "28-Jul"
```

```r
format(Sys.Date(), "%Y")
```

```
## [1] "2017"
```

R can even use the seq command to generate strings of dates, which can be helpful when cleaning data.

```r
dates <- seq(as.Date("2015-01-01"), to = as.Date("2015-12-31"), by = "month")
print(dates)
```

```
##  [1] "2015-01-01" "2015-02-01" "2015-03-01" "2015-04-01" "2015-05-01"
##  [6] "2015-06-01" "2015-07-01" "2015-08-01" "2015-09-01" "2015-10-01"
## [11] "2015-11-01" "2015-12-01"
```

```r
mean(dates)
```

```
## [1] "2015-06-16"
```

```r
max(dates)
```

```
## [1] "2015-12-01"
```

In-class exercise:

Use the seq() command to generate a vector of every month-end date in 2010.

Rather technical notes:
In the case of two digit years, R (currently) assumes that years 00-68 are 2000 - 2068, and years 69-99 are 1969 - 1999.

R can even import from Excel, with Excel's wierd 5-digit dates.

```r
# From Windows Excel
as.Date(30829, origin = "1899-12-30")
```

```
## [1] "1984-05-27"
```

```r
# From iOS Excel
as.Date(29367, origin = "1904-01-01")
```

```
## [1] "1984-05-27"
```

Using lubridate to handle dates

```r
# Lubridate introduction

# Date creation functions:
# ymd()
# dmy()
# mdy()
```

```
banks <- banks %>%
              mutate(date = ymd(date))
```

Lubridate (like dplyr) is just meant to make working with objects in R as intuitive and straightforward as possible. The date creation functions of lubridate do what as.Date() does, but the formatting conditions are not buried within an argument in the function; rather, each format type is denoted by the function name itself. The following date accessor functions in lubridate will accomplish the same thing as format(), and also outperform it in some ways by providing accessibility to a wider array of representations of time units.

```
# 'Accessor' functions:
# year()
# month()
# mday()
# yday()
# wday()

banks <- banks %>%
              mutate(monthonly = month(date),
                     yearonly  = year(date))
```

Now we know and, perhaps more importantly, can reference the month and the year belonging to each observation.
* Which functions from dplyr could we next use to calculate our desired monthly statistic(s)?

2. Strings have extra, unneeded information

Text patterns, splitting, and extracting

The three major charter types for depository institutions are:

1. commercial banks,
2. savings/thrift banks, and
3. credit unions.

We might believe banks behave differently based on their charter type for many reasons; different regulatatory requirements and different targeted customer base might be two of the most compelling. Therefore, we will want to make sure this measurement is reasonably coded in our dataset.

```
# Let's see what kinds of charters we have in the variable 'type'
unique(banks$type)
```

```
## [1] "Commercial"    "Credit Union" "Savings"      "Saving"
## [5] "Savigns"       "savins"       "savin"        "savngi"
```

It looks like some savings banks were entered as "Saving" and some as "Savings", as well as a number of misspellings of these. We will want all of these to be the same exact string because they all represent only one of the possible charter types. Doing so will allow us to correctly identify observations on savings banks. This is likely to come in handy anytime we want to make use of subgroups in our data. You learned previously about using group_by() to declare subgroups in your data - that can only be effective for us if the unique values by which we group_by() make sense. We could naively go through each unique misspelling and reassign its value, or, more efficiently, we can use textual patterns to identify and select all of the misspellings at once. To achieve this end we will make use of the stringr package.

Using stringr to handle character strings

Every stringr function begins with the prefix str_, str being short for string. RStudio's tab-complete feature will therefore let you browse all stringr functions at once so you can quickly find the function most suited to handle your particular coding challenge. As the names suggest, the stringr functions are a toolbox of string manipulators to be used at your disposal whenever you need work done on character objects. As you see for yourself, many of these string functions have to do with locating patterns in text.

```
# str_detect returns a logical vector for where the pattern was found
str_detect(string = c("a", "b", "c"), pattern = "b")
```

```
## [1] FALSE  TRUE FALSE
```

```
# str_which returns the indices where the pattern was found
str_which(string = c("a", "b", "c"), pattern = "b")
```

```
## [1] 2
```

```
# Replaces patterns within text strings
str_replace("abc", pattern = "a", replacement = "c")
```

```
## [1] "cbc"
```

We can use str_detect() to filter to observations that somewhat look like Savings. Once we have identified these elements, we can then reassign all of those to the same phrase. First, we will need an introduction to str_to_lower().

```
# str_to_lower
str_to_lower(c("A", "B", "C"))
```

```
## [1] "a" "b" "c"
```

```
str_to_lower(c("Lower", "Case"))
```

```
## [1] "lower" "case"
```

```
# Alternatively, there is a str_to_upper
str_to_upper((c("Upper", "Case")))
```

```
## [1] "UPPER" "CASE"
```

This is useful in our case since there are multiple variations of Savings both with and without upper case characters. To make our job of detecting these variations simpler, we will use a version of the variable 'type' where all characters are in the same case.

```
type_lower <- str_to_lower(banks$type)

#Using str_detect to filter rows
banks[str_detect(type_lower, "sav"), "type"] <- "Savings"

# Check unique values of 'type' again to see if we need to address more cases of
# misspellings
unique(banks$type)
```

```
## [1] "Commercial"    "Credit Union" "Savings"
```

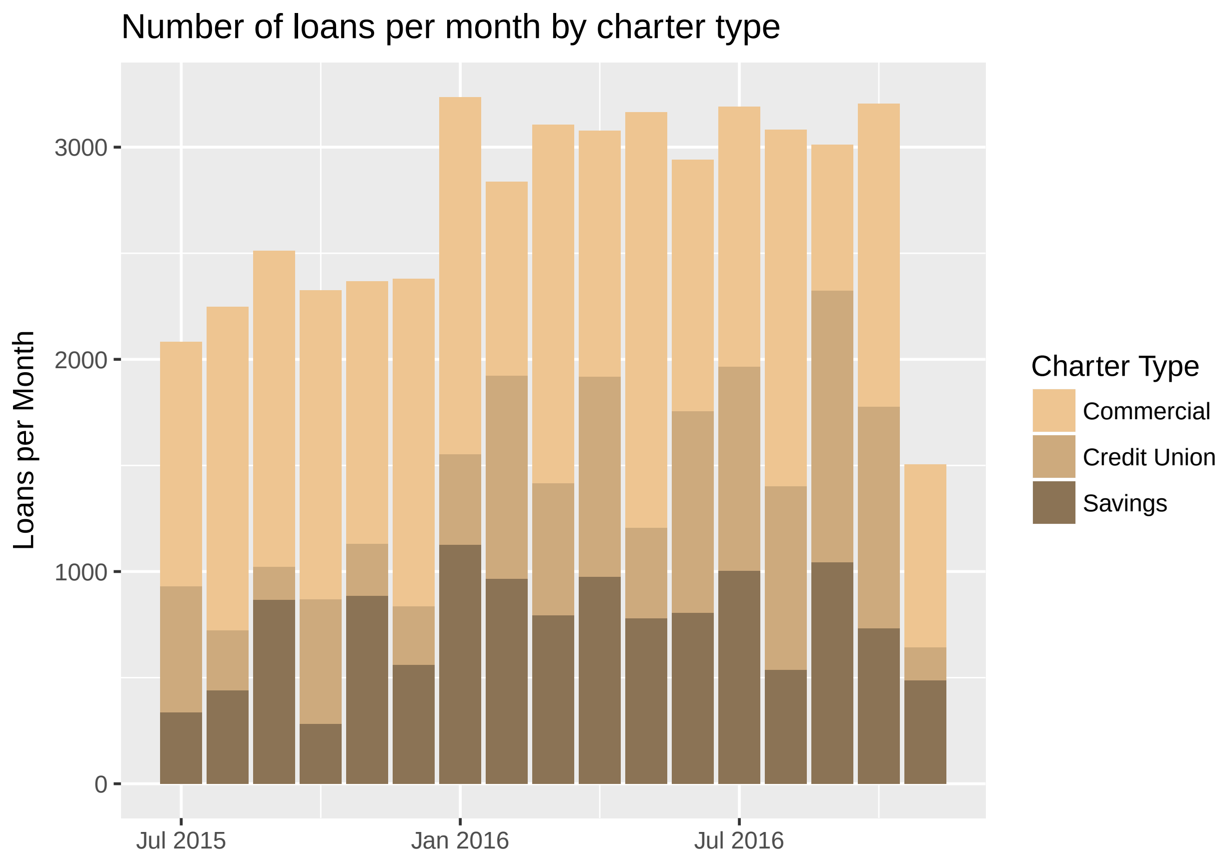
```
# No longer need type_lower - can remove it to tidy up the environment
rm(type_lower)
```

Now that the typos are gone, what are some tasks we can do that we could not before? Can we extract information about meaningful subgroups in our dataset?

```
CharterSummary <- banks %>%
                    group_by(yearonly, monthonly, type) %>%
                    summarize(monthlyCount = sum(amount))
CharterSummary
```

```
## # A tibble: 51 x 4
## # Groups:   yearonly, monthonly [?]
##    yearonly monthonly         type monthlyCount
##       <dbl>     <dbl>        <chr>        <int>
##  1     2015         7   Commercial         1152
##  2     2015         7 Credit Union          595
##  3     2015         7      Savings          336
##  4     2015         8   Commercial         1524
##  5     2015         8 Credit Union          285
##  6     2015         8      Savings          439
##  7     2015         9   Commercial         1490
##  8     2015         9 Credit Union          155
##  9     2015         9      Savings          867
## 10     2015        10   Commercial         1457
## # ... with 41 more rows
```

So for each true, unique type of bank in our dataset we have a monthly count of issued loans. We could have instead chosen, and still can choose, to look at quarterly totals, yearly totals, daily totals, etc. Can you think of reasons why we aggregated our data up to the month level?



In Class Exercise:

Convert the following vector, numbers, to numeric type.

```r
numbers <- c("2,100", "3,250,000")
```

The next task concerns yet another measurement of interest: proportion of private to government loans. In order to even be able to calculate a proportion we would need variables that correctly identify private and government loans.
* Do we have variables that accomplish this yet?

```r
# Let's take a look at loanID
unique(banks$loanID)
```

```
## [1] "P1" "P2" "P4" "G2" "G1" "G3" "P3"
```

Every value in loanID begins with either a "P" or a "G". "P" is shorthand for private, and so it denotes private loans, and "G" is shorthand for government, so observations with loanID "G…" are government loans. This is pretty close to the measurement we want, but not quite. Our measurement needs to be generalised enough so that we are just comparing private and government across observations. What we really need are just the P's and G's. Introducing str_sub():

```r
# To completely define a substring, you always only need three things:
## 1. the full string,
## 2. the starting position of the substring, and
## 3. the ending position of the substring.
str_sub("abcd", start = 1, end = 3) # Extract characters from a text string
```

```
## [1] "abc"
```

```r
str_sub("abcd", start = 1, end = 2)
```

```
## [1] "ab"
```

To split a string apart into multiple pieces we would use str_split().

```r
str_split("abcd", pattern = "c") # Splits a text string whenever the pattern argument is found
```

```
## [[1]]
## [1] "ab" "d"
```

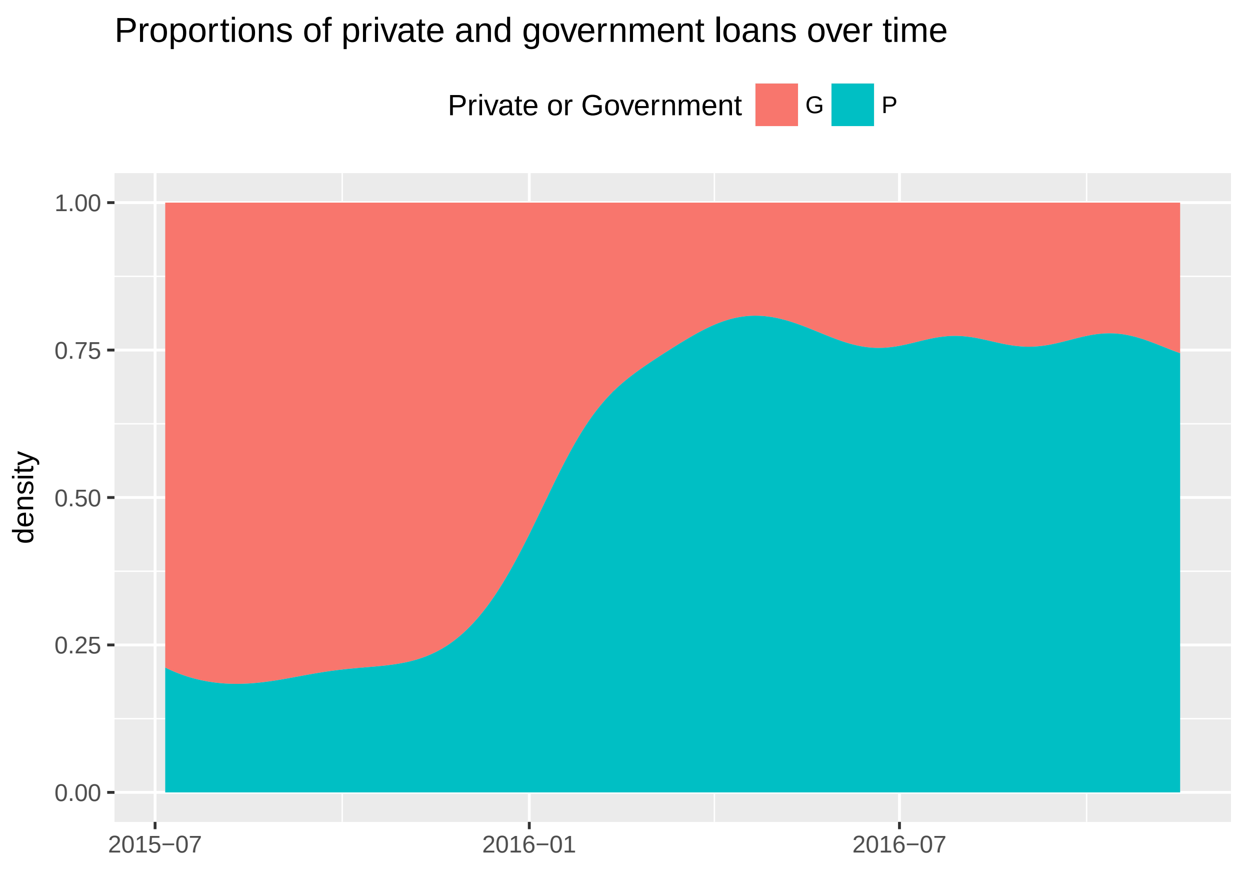
```r
str_split("keep_these_parts", pattern = "_")
```

```
## [[1]]
## [1] "keep"  "these" "parts"
```

- Seeing these two stringr functions in action, which one could we use to arrive at our goal of extracting the P's and G's from loanID?

```r
# We can use str_sub() to extract the P's and G's in loanID
banks <- banks %>%
              mutate(loantype = str_sub(loanID, start = 1, end = 1))

ggplot(banks, aes(x = date,
                  fill = as.factor(loantype))) +
  geom_density(position = "fill", color = NA) +
  scale_x_date() +
  labs(fill = "Private or Government", x = NULL,
       title = "Proportions of private and government loans over time") +
  theme(legend.position = "top")
```



3. Information in a table is difficult to extract

We will be using tidyr's two main functions: spread() and gather(). The former helps you deal with rows that are not complete observations and the latter with columns that are not variables. To demonstrate these methods of tidying data, we will be using the two datasets stocks_l and stocks_w.

```
stocks_l
```

```
##   Stock Year Price
## 1  AAPL 2007   400
## 2  AAPL 2008   450
## 3  AAPL 2009   500
## 4  AMZN 2007   200
## 5  AMZN 2008   150
## 6  AMZN 2009   200
## 7  ADBE 2007    30
## 8  ADBE 2008    10
## 9  ADBE 2009    40
```

```
stocks_w
```

```
##   Stock 2007 2008 2009
## 1  AAPL  400  450  500
## 2  ADBE   30   10   40
## 3  AMZN  200  150  200
```

Believe it or not the same exact information is stored in these two datasets. The "form" that stocks_l takes is often referred to as long-form data and stocks_w wide-form data. Advantages exist for both forms:

- Long
  - Conceptually clear - you can quickly recognize what constitutes a unique observation in the data.
  - Great for grouping - if you want color, shape, fill, etc. dimensions in your ggplots this is the form you want.
- Wide
  - Extremely common way to store data - the people storing data are typically those that enter data. Wide form usually makes entering data easier.

We can go from one to the other fairly easily in R thanks to packages like tidyr and reshape2.

```r
library(tidyr)

stocks_w %>%
  gather(`2007`, `2008`, `2009`, key = "Year", value = "Price")
```

```
##   Stock Year Price
## 1  AAPL 2007   400
## 2  ADBE 2007    30
## 3  AMZN 2007   200
## 4  AAPL 2008   450
## 5  ADBE 2008    10
## 6  AMZN 2008   150
## 7  AAPL 2009   500
## 8  ADBE 2009    40
## 9  AMZN 2009   200
```

Besides some rearrangement of rows, we put stocks_w through a gather() function and got an output exactly like stocks_l.

Think of spread() and gather() as inverses: one goes wide-to-long and the other long-to-wide (generally). Here we will spread stocks_l, although typically there is little reason to spread a tidy dataset like stocks_l.

```r
stocks_l %>%
  spread(key = Year, value = Price)
```

```
##   Stock 2007 2008 2009
## 1  AAPL  400  450  500
## 2  ADBE   30   10   40
## 3  AMZN  200  150  200
```

HOMEWORK