

# Chapter 3: Data Wrangling

Mark Andrews

## Contents

<b>Data wrangling tools in R</b>	<b>2</b>
<b>Reading text file data into a data frame</b>	<b>2</b>
<b>Manipulating data frames using dplyr</b>	<b>3</b>
Selecting variables with <code>select</code> . . . . .	3
Renaming variables with <code>rename</code> . . . . .	10
Selecting observations with <code>slice</code> and <code>filter</code> . . . . .	12
Changing variables and values with <code>mutate</code> . . . . .	17
Sorting observations with <code>arrange</code> . . . . .	24
Subsampling data frames . . . . .	25
Reducing data with <code>summarize</code> and <code>group_by</code> . . . . .	26
<b>The <code>%&gt;%</code> operator</b>	<b>29</b>
<b>Combining data frames</b>	<b>32</b>
Combining data frames with <code>binds</code> . . . . .	32
Combining data frames by <code>joins</code> . . . . .	33
Combining data frames by set operations . . . . .	36
<b>Reshaping with <code>pivot_longer</code> and <code>pivot_wider</code></b>	<b>37</b>
<b>References</b>	<b>40</b>

Traditional statistics textbooks and courses routinely assume that the data is ready for analysis. Their starting point for any analysis is usually a neat table of rows and columns, with all and only the relevant variables, each with meaningful names, and often accompanied with a useful description or summary of what each variable signifies or measures. In reality, on the other hand, the starting point of any analysis is almost always a very messy, unstructured or ill-formatted data set, or even multiple separate data sets, that must first be cleaned up and modified before any further analysis can begin. Throughout this book, we will use the term *data wrangling* to describe the process of taking data in its unstructured, messy, or complicated original form and converting it into a clean and tidy format that allows data exploration, visualization, and eventually statistical modelling and analysis to proceed efficiently and relatively effortlessly. Other terms for data wrangling include *data munging*, *data cleaning*, *data pre-processing*, *data preparation*, and so on.

The central role of data wrangling in any type of data analysis should not be underestimated. Part of lore of modern data science is the belief that up 80% of all data science activities involves data wrangling (see, for example, “For Big-Data Scientists, ‘Janitor Work’ Is Key Hurdle to Insights” 2014), and this 80% figure is backed up surveys of what data scientists do (see, for example, “CrowdFlower 2016 Data Science Report” 2016; “CrowdFlower 2017 Data Scientist Report” 2017). Even if this number is not accurate, data wrangling is a necessary and potentially very time consuming and laborious activity for any data analysis. As such, developing data wrangling skills is essential for doing data analysis efficiently.

## Data wrangling tools in R

There are many tools in R for doing data wrangling. Here, we will focus on a core set of inter-related `tidyverse` tools. These include the commands available in the `dplyr` package, particularly its so-called *verbs* such as the following.

- `select`
- `rename`
- `slice`
- `filter`
- `mutate`
- `arrange`
- `group_by`
- `summarize`

In addition, `dplyr` provides tools for merging and joining data sets such as the following:

- `inner_join`
- `left_join`
- `right_join`
- `full_join`

Next, there are the tools in the `tidyr` package, particularly the following:

- `pivot_longer`
- `pivot_wider`

These and other tools can then be combined together using the `%>%` pipe operator for efficient data analysis *pipelines*.

Most of these tools can be loaded into R by loading the `tidyverse` package of packages.

```
library(tidyverse)
```

## Reading text file data into a data frame

In principle, raw data can exist in any format in any type file. In practice, it is common to have data in a roughly rectangular format, i.e. with rows and columns, either in text files such as `.csv`, `.tsv`, or `.txt` files. The `readr` package, which is loaded when we load `tidyverse`, allows us to efficiently import data that are in these files. It has many commands for importing data in many different text file formats. The most commonly used include

- `read_csv` for files where the values on each line are separated by commas
- `read_tsv` for files where the values are separated by tabs
- `read_delim` for files where the values are separated by arbitrary delimiters such as `'|'`, `':'`, `','`, etc. Both `read_csv` and `read_tsv` are special cases of the more general `read_delim` command.
- `read_table` for files where the values are separated by one or more, and possibly inconsistently many, whitespaces.

These commands usually read from files stored locally on the computer on which R is running. For example, if we have a `.csv` file named `data.csv` that is inside a directory called `data` that was in our working directory, we would read this by default as follows.

```
read_csv('data/data.csv')
```

However, these commands also can read from files on the internet. In this case, we provide a url for the file. These commands can also read compressed files if they are compressed in the `.xz`, `.bz2`, `.gz`, or `.zip` compression formats.

As an example data set, we will use the data contained in the file `blp-trials-short.txt`. We will read it in to a data frame named `blp_df` as follows:

```
blp_df <- read_csv("data/blp-trials-short.txt")
blp_df
#> # A tibble: 1,000 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      20 N   staud N   977 511 977
#> 2      9 N   dinbuss N   565 765 565
#> 3     47 N   snilling N   562 496 562
#> 4    103 N   gancens N   572 656 572
#> 5     45 W   filled W   659 981 659
#> 6     73 W   journals W   538 1505 538
#> 7     24 W   apache W   626 546 626
#> 8     11 W   flake W   566 717 566
#> 9     32 W   reliefs W   922 1471 922
#> 10    96 N   sarves N   555 806 555
#> # ... with 990 more rows
```

We can use the `dplyr` command `glimpse` to look at resulting data frame.

```
glimpse(blp_df)
#> Rows: 1,000
#> Columns: 7
#> $ participant <dbl> 20, 9, 47, 103, 45, 73, 24, 11, 32, 96, 82, 37, 52, 96, ...
#> $ lex <chr> "N", "N", "N", "N", "W", "W", "W", "W", "W", "N", "W", ...
#> $ spell <chr> "staud", "dinbuss", "snilling", "gancens", "filled", "j...
#> $ resp <chr> "N", "N", "N", "N", "W", "W", "W", "W", "W", "N", "W", ...
#> $ rt <dbl> 977, 565, 562, 572, 659, 538, 626, 566, 922, 555, 657, ...
#> $ prev.rt <dbl> 511, 765, 496, 656, 981, 1505, 546, 717, 1471, 806, 728...
#> $ rt.raw <dbl> 977, 565, 562, 572, 659, 538, 626, 566, 922, 555, 657, ...
```

As we can see, there are 1000 rows and 7 variables. This data frame gives the trial by trial results from a type of cognitive psychology experiment known as a *lexical decision task*. In a lexical decision task, participants are shown a string of characters and they have to indicate, with a key press, whether that string of characters is a word in their language. On each row of the data frame, among other things, we have an identifier of the participant, what string of characters they were shown, what key the pressed, what their reaction time was, and so on.

## Manipulating data frames using `dplyr`

The `dplyr` package provides a set versatile inter-related commands for manipulating data frames. Chief amongst these commands are `dplyr`'s *verbs* listed above. Here, we will look at each one.

### Selecting variables with `select`

In our `blp_df` data frames we have 7 variables. Let's say, as is often the case when processing raw data, that we only need some of these. The `dplyr` command `select` allows us to select those we want. For example, if we just want the participant's id, whether the displayed string was a English word or not, what their key press response was, what their reaction time was, then we would do the following.

```
select(blp_df, participant, lex, resp, rt)
#> # A tibble: 1,000 x 4
#>   participant lex resp rt
#>   <dbl> <chr> <chr> <dbl>
```

```

#> 1      20 N      N      977
#> 2       9 N      N      565
#> 3      47 N      N      562
#> 4     103 N      N      572
#> 5      45 W      W      659
#> 6      73 W      W      538
#> 7      24 W      W      626
#> 8      11 W      W      566
#> 9      32 W      W      922
#> 10     96 N      N      555
#> # ... with 990 more rows

```

Importantly, `select` returns a *new* data frame with the selected variables. In other words, the original `blp` data frame is still left fully intact. This feature of returning a new data frame and not altering the original data frame is true of all of the `dplyr` verbs and many other wrangling commands that we'll meet below.

We can select a range of variables by specifying the first and last variables in the range with a `:` between them as follows.

```

select(blp_df, spell:prev.rt)
#> # A tibble: 1,000 x 4
#>   spell      resp      rt prev.rt
#>   <chr>    <chr> <dbl> <dbl>
#> 1 staud      N      977    511
#> 2 dinbuss    N      565    765
#> 3 snilling  N      562    496
#> 4 gancens    N      572    656
#> 5 filled    W      659    981
#> 6 journals  W      538   1505
#> 7 apache    W      626    546
#> 8 flake     W      566    717
#> 9 reliefs   W      922   1471
#> 10 sarves   N      555    806
#> # ... with 990 more rows

```

We can also select a range of variables using indices as in the following example.

```

select(blp_df, 2:5) # columns 2 to 5
#> # A tibble: 1,000 x 4
#>   lex spell      resp      rt
#>   <chr> <chr>    <chr> <dbl>
#> 1 N     staud      N      977
#> 2 N     dinbuss    N      565
#> 3 N     snilling  N      562
#> 4 N     gancens    N      572
#> 5 W     filled    W      659
#> 6 W     journals  W      538
#> 7 W     apache    W      626
#> 8 W     flake     W      566
#> 9 W     reliefs   W      922
#> 10 N    sarves    N      555
#> # ... with 990 more rows

```

We can select variables according to the character or characters that they begin with. For example, we select all variables that begin with `p` as follows.

```

select(blp_df, starts_with('p'))

```

```

#> # A tibble: 1,000 x 2
#>   participant prev.rt
#>   <dbl> <dbl>
#> 1      20     511
#> 2       9     765
#> 3      47     496
#> 4     103     656
#> 5      45     981
#> 6      73    1505
#> 7      24     546
#> 8      11     717
#> 9      32    1471
#> 10     96     806
#> # ... with 990 more rows

```

Or we can select variables by the characters they end with.

```

select(blp_df, ends_with('t'))
#> # A tibble: 1,000 x 3
#>   participant    rt prev.rt
#>   <dbl> <dbl> <dbl>
#> 1      20   977     511
#> 2       9   565     765
#> 3      47   562     496
#> 4     103   572     656
#> 5      45   659     981
#> 6      73   538    1505
#> 7      24   626     546
#> 8      11   566     717
#> 9      32   922    1471
#> 10     96   555     806
#> # ... with 990 more rows

```

We can select variables that contain a certain set of characters in any position. For example, the following selects variables whose names contain the string `rt`.

```

select(blp_df, contains('rt'))
#> # A tibble: 1,000 x 4
#>   participant    rt prev.rt rt.raw
#>   <dbl> <dbl> <dbl> <dbl>
#> 1      20   977     511   977
#> 2       9   565     765   565
#> 3      47   562     496   562
#> 4     103   572     656   572
#> 5      45   659     981   659
#> 6      73   538    1505   538
#> 7      24   626     546   626
#> 8      11   566     717   566
#> 9      32   922    1471   922
#> 10     96   555     806   555
#> # ... with 990 more rows

```

The previous example selected the variable `participant` because it contained the word `rt`. However, if we had wanted to select only those variables that contained `rt` where it clearly meant reaction time, we could use a *regular expression* match. For example, the regular expression `^rt|rt$` will match the `rt` if it begins or ends a string. Therefore, we can select the variables that contain `rt`, where the string `rt` means reaction

time, as follows.

```
select(blp_df, matches('^rt|rt$'))
#> # A tibble: 1,000 x 3
#>   rt prev.rt rt.raw
#>   <dbl> <dbl> <dbl>
#> 1  977    511   977
#> 2  565    765   565
#> 3  562    496   562
#> 4  572    656   572
#> 5  659    981   659
#> 6  538   1505   538
#> 7  626    546   626
#> 8  566    717   566
#> 9  922   1471   922
#> 10 555    806   555
#> # ... with 990 more rows
```

*Removing variables:* We can use `select` to *remove* variables as well as select them. To remove a variable, we precede its name with a minus sign.

```
select(blp_df, -participant) # remove `participant`
#> # A tibble: 1,000 x 6
#>   lex spell resp rt prev.rt rt.raw
#>   <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1 N      staud  N      977    511   977
#> 2 N      dinbuss N      565    765   565
#> 3 N      snilling N      562    496   562
#> 4 N      gancens N      572    656   572
#> 5 W      filled W      659    981   659
#> 6 W      journals W      538   1505   538
#> 7 W      apache W      626    546   626
#> 8 W      flake W      566    717   566
#> 9 W      reliefs W      922   1471   922
#> 10 N     sarves N      555    806   555
#> # ... with 990 more rows
```

Just as we selected ranges or sets of variables above, we can remove them by preceding their selection functions with minus signs. For example, to remove variables indexed 2 to 6, we would do the following.

```
select(blp_df, -(2:6))
#> # A tibble: 1,000 x 2
#>   participant rt.raw
#>   <dbl> <dbl>
#> 1      20   977
#> 2       9   565
#> 3      47   562
#> 4     103   572
#> 5      45   659
#> 6      73   538
#> 7      24   626
#> 8      11   566
#> 9      32   922
#> 10     96   555
#> # ... with 990 more rows
```

Or, as another example, we can remove the variables that contain the string `rt` as follows.

```

select(blp_df, -contains('rt'))
#> # A tibble: 1,000 x 3
#>   lex  spell  resp
#>   <chr> <chr>  <chr>
#> 1 N     staud   N
#> 2 N     dinbuss N
#> 3 N     snilling N
#> 4 N     gancens N
#> 5 W     filled  W
#> 6 W     journals W
#> 7 W     apache  W
#> 8 W     flake   W
#> 9 W     reliefs W
#> 10 N    sarves  N
#> # ... with 990 more rows

```

*Reordering variables:* When we select variables with `select`, we control their order in the resulting data frame. For example, if we select `spell`, `participant`, `res`, the resulting data frame will have them in their selected order.

```

select(blp_df, spell, participant, resp)
#> # A tibble: 1,000 x 3
#>   spell      participant resp
#>   <chr>          <dbl> <chr>
#> 1 staud             20 N
#> 2 dinbuss            9 N
#> 3 snilling          47 N
#> 4 gancens          103 N
#> 5 filled            45 W
#> 6 journals           73 W
#> 7 apache            24 W
#> 8 flake             11 W
#> 9 reliefs           32 W
#> 10 sarves           96 N
#> # ... with 990 more rows

```

However, clearly the resulting data frame only returned those variables that we selected. We can, however, include all remaining variables after those we explicitly selected by using `everything()` as follows.

```

select(blp_df, spell, participant, resp, everything())
#> # A tibble: 1,000 x 7
#>   spell      participant resp lex      rt prev.rt rt.raw
#>   <chr>          <dbl> <chr> <chr> <dbl>  <dbl> <dbl>
#> 1 staud             20 N     N     977    511    977
#> 2 dinbuss            9 N     N     565    765    565
#> 3 snilling          47 N     N     562    496    562
#> 4 gancens          103 N     N     572    656    572
#> 5 filled            45 W     W     659    981    659
#> 6 journals           73 W     W     538   1505    538
#> 7 apache            24 W     W     626    546    626
#> 8 flake             11 W     W     566    717    566
#> 9 reliefs           32 W     W     922   1471    922
#> 10 sarves           96 N     N     555    806    555
#> # ... with 990 more rows

```

We can also use `everything` to move some variables to the start of the list, and some to the end, and have

the remaining variables in the middle. For example, we can move `resp` to the start of the list of variables, move to `participant` to the end, and then have everything else in between as follows.

```
select(blp_df, resp, everything(), -participant, participant)
#> # A tibble: 1,000 x 7
#>   resp lex spell rt prev.rt rt.raw participant
#>   <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl>
#> 1 N N staud 977 511 977 20
#> 2 N N dinbuss 565 765 565 9
#> 3 N N snilling 562 496 562 47
#> 4 N N gancens 572 656 572 103
#> 5 W W filled 659 981 659 45
#> 6 W W journals 538 1505 538 73
#> 7 W W apache 626 546 626 24
#> 8 W W flake 566 717 566 11
#> 9 W W reliefs 922 1471 922 32
#> 10 N N sarves 555 806 555 96
#> # ... with 990 more rows
```

In this example, we essentially move `resp` to the front of the list, followed by all remaining variables. Then we remove `participant` by `-participant` and then re-insert it at the end of the list of the remaining variables.

*Selecting by condition with `select_if`*: Thus far, we have selected variables according to properties of their names or by their indices. The `select_if` function is a powerful function that allows us to select variables according to properties of their values. For example, the function `is.character` will verify whether a vector is a character vector or not, and `is.numeric` will verify if a vector is a numeric vector, as in the following.

```
x <- c(1, 42, 3)
y <- c('good', 'dogs', 'brent')
is.numeric(x)
#> [1] TRUE
is.numeric(y)
#> [1] FALSE
is.character(x)
#> [1] FALSE
is.character(y)
#> [1] TRUE
```

By passing the function `is.character` to select the variables that are character vectors as follows.

```
select_if(blp_df, is.character)
#> # A tibble: 1,000 x 3
#>   lex spell resp
#>   <chr> <chr> <chr>
#> 1 N staud N
#> 2 N dinbuss N
#> 3 N snilling N
#> 4 N gancens N
#> 5 W filled W
#> 6 W journals W
#> 7 W apache W
#> 8 W flake W
#> 9 W reliefs W
#> 10 N sarves N
#> # ... with 990 more rows
```



Note that in this command, we pass the function itself, i.e. `is.numeric`. We do not use the function call, i.e. `is.numeric()`. In the following example, we select the numeric variables in `blp`.

```
select_if(blp_df, is.numeric)
#> # A tibble: 1,000 x 4
#>   participant    rt prev.rt rt.raw
#>   <dbl> <dbl>   <dbl> <dbl>
#> 1         20   977     511   977
#> 2          9   565     765   565
#> 3         47   562     496   562
#> 4        103   572     656   572
#> 5         45   659     981   659
#> 6         73   538    1505   538
#> 7         24   626     546   626
#> 8         11   566     717   566
#> 9         32   922    1471   922
#> 10        96   555     806   555
#> # ... with 990 more rows
```

We can use custom functions with `select_if`. In the Chapter 2, we briefly described how to create custom functions in R. This is a topic to which we will return in more depth Chapter 6. Now, and throughout the remainder of this chapter, we will create some custom functions to use with data wrangling, but we will not describe delve too deep into the details of how they work.

As an example, the following function will return `TRUE` if the variable is a numeric variable with a mean that is less than 700.

```
has_low_mean <- function(x){
  is.numeric(x) && (mean(x, na.rm = T) < 700)
}
```

Now, we can select variables that meet this criterion as follows.

```
select_if(blp_df, has_low_mean)
#> # A tibble: 1,000 x 3
#>   participant    rt prev.rt
#>   <dbl> <dbl>   <dbl>
#> 1         20   977     511
#> 2          9   565     765
#> 3         47   562     496
#> 4        103   572     656
#> 5         45   659     981
#> 6         73   538    1505
#> 7         24   626     546
#> 8         11   566     717
#> 9         32   922    1471
#> 10        96   555     806
#> # ... with 990 more rows
```

We can also use an *anonymous* function within `select_if`. An anonymous function is a function without a name, and its use is primarily for situations where functions are used in a one-off manner, and so there is no need to save them. As an example, the anonymous version of `has_low_mean` is simply the following.

```
function(x){ is.numeric(x) && (mean(x, na.rm = T) < 700) }
```

We can put this anonymous function inside `select_if` as follows.

```
select_if(blp_df, function(x){ is.numeric(x) && (mean(x, na.rm = T) < 700) })
#> # A tibble: 1,000 x 3
```

```

#>   participant    rt prev.rt
#>   <dbl> <dbl>  <dbl>
#> 1         20   977    511
#> 2          9   565    765
#> 3         47   562    496
#> 4        103   572    656
#> 5         45   659    981
#> 6         73   538   1505
#> 7         24   626    546
#> 8         11   566    717
#> 9         32   922   1471
#> 10        96   555    806
#> # ... with 990 more rows

```

We can make a less verbose version of this anonymous function using a syntactic shortcut that is part of the `purrr` package, which is loaded when we load `tidyverse`, as follows.

```

select_if(bl_df, ~is.numeric(.) && (mean(., na.rm = T) < 700))
#> # A tibble: 1,000 x 3
#>   participant    rt prev.rt
#>   <dbl> <dbl>  <dbl>
#> 1         20   977    511
#> 2          9   565    765
#> 3         47   562    496
#> 4        103   572    656
#> 5         45   659    981
#> 6         73   538   1505
#> 7         24   626    546
#> 8         11   566    717
#> 9         32   922   1471
#> 10        96   555    806
#> # ... with 990 more rows

```

## Renaming variables with `rename`

When we select individual variables with `select`, we can rename them too, as in the following example.

```

select(bl_df, subject=participant, reaction_time=rt)
#> # A tibble: 1,000 x 2
#>   subject reaction_time
#>   <dbl>         <dbl>
#> 1      20           977
#> 2       9           565
#> 3      47           562
#> 4     103           572
#> 5      45           659
#> 6      73           538
#> 7      24           626
#> 8      11           566
#> 9      32           922
#> 10     96           555
#> # ... with 990 more rows

```

While this is useful, the data frame that is returned just contains the selected variables. If we want to rename some variables, and get a data frame with all variables, including the renamed ones, we should use `rename`.

```

rename(bl_df, subject=participant, reaction_time=rt)

```

```

#> # A tibble: 1,000 x 7
#>   subject lex  spell  resp reaction_time prev.rt rt.raw
#>   <dbl> <chr> <chr>  <chr>      <dbl>    <dbl> <dbl>
#> 1     20 N   staud    N         977      511  977
#> 2      9 N   dinbuss  N         565      765  565
#> 3     47 N   snilling N         562      496  562
#> 4    103 N   gancens  N         572      656  572
#> 5     45 W   filled   W         659      981  659
#> 6     73 W   journals W         538     1505  538
#> 7     24 W   apache   W         626      546  626
#> 8     11 W   flake    W         566      717  566
#> 9     32 W   reliefs  W         922     1471  922
#> 10    96 N   sarves   N         555      806  555
#> # ... with 990 more rows

```

Useful variants of `rename` include `rename_all`, `rename_at`, and `rename_if`. The `rename_all` function allows us to, as the name implies, rename all the variables using some renaming function, i.e., a function that takes a string as input and returns another as output. As an example of such a function, here is a `purrr` style anonymous function, using the `str_replace_all` function from the `stringr` package, that replaces any dot in the variable name with an underscore.

```

rename_all(blp_df, ~str_replace_all(., '\\.', '_'))
#> # A tibble: 1,000 x 7
#>   participant lex  spell  resp  rt prev_rt rt_raw
#>   <dbl> <chr> <chr>  <chr> <dbl> <dbl> <dbl>
#> 1      20 N   staud    N     977    511  977
#> 2       9 N   dinbuss  N     565    765  565
#> 3      47 N   snilling N     562    496  562
#> 4     103 N   gancens  N     572    656  572
#> 5      45 W   filled   W     659    981  659
#> 6      73 W   journals W     538   1505  538
#> 7      24 W   apache   W     626    546  626
#> 8      11 W   flake    W     566    717  566
#> 9      32 W   reliefs  W     922   1471  922
#> 10     96 N   sarves   N     555    806  555
#> # ... with 990 more rows

```

In this example, because `str_replace_all` uses regular expressions for text pattern matching, and in a regular expression a `.` character means “any character”, we have to use `\\.` to refer to a literal dot.

The `rename_at` function allows us to select certain variables, and then apply a renaming function just to these selected variables. We can use selection functions like `contains` or `matches` that we used above, but it is necessary to surround these functions with the `vars` function. In the following example, we select all variables whose names contain `rt` at their start or end, and then replace their occurrences of `rt` with `reaction_time`.

```

rename_at(blp_df,
  vars(matches('^rt|rt$')),
  ~str_replace_all(., 'rt', 'reaction_time'))
#> # A tibble: 1,000 x 7
#>   participant lex  spell resp  reaction_time prev.reaction_t~ reaction_time.r~
#>   <dbl> <chr> <chr> <chr>      <dbl>          <dbl>          <dbl>
#> 1      20 N   staud  N         977            511            977
#> 2       9 N   dinb~  N         565            765            565
#> 3      47 N   snil~  N         562            496            562
#> 4     103 N   ganc~  N         572            656            572
#> 5      45 W   fill~  W         659            981            659

```

```

#> 6          73 W    jour~ W          538          1505          538
#> 7          24 W    apac~ W          626           546          626
#> 8          11 W    flake W          566           717          566
#> 9          32 W    reli~ W          922          1471          922
#> 10         96 N    sarv~ N          555           806          555
#> # ... with 990 more rows

```

Similarly to how we used `select_if`, `rename_if` can be used to rename variables whose values match certain criteria. For example, if we wanted to capitalize the names of those variables that are character variables, we could do the following.

```

rename_if(blp_df, is.character, str_to_upper)
#> # A tibble: 1,000 x 7
#>   participant LEX  SPELL  RESP    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>  <chr> <dbl> <dbl> <dbl>
#> 1         20 N    staud  N     977    511    977
#> 2          9 N    dinbuss N     565    765    565
#> 3         47 N    snilling N     562    496    562
#> 4        103 N    gancens N     572    656    572
#> 5         45 W    filled  W     659    981    659
#> 6         73 W    journals W     538   1505    538
#> 7         24 W    apache  W     626    546    626
#> 8         11 W    flake   W     566    717    566
#> 9         32 W    reliefs W     922   1471    922
#> 10        96 N    sarves  N     555    806    555
#> # ... with 990 more rows

```

In this example, we use the `str_to_upper` from the package `stringr`, which is also loaded by `tidyverse`, to convert the names of the selected variables to uppercase.

## Selecting observations with `slice` and `filter`

With `select` and `rename`, we were selecting or removing variables. The commands `slice` and `filter` allow us to select or remove observations. We use `slice` to select observations by their indices. For example, to select rows 10, 20, 50, 100, 500, we would simply do the following.

```

slice(blp_df, c(10, 20, 50, 100, 500))
#> # A tibble: 5 x 7
#>   participant lex  spell  resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1         96 N    sarves N     555    806    555
#> 2         46 W    mirage W     778    571    778
#> 3         72 N    gright N     430    675    430
#> 4          3 W    gleam  W     361    370    361
#> 5         92 W    coaxes W     699    990    699

```

Given that, for example, `10:100` would list the integers 10 to 100 inclusive, we can select just these observations as follows.

```

slice(blp_df, 10:100)
#> # A tibble: 91 x 7
#>   participant lex  spell  resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>  <chr> <dbl> <dbl> <dbl>
#> 1         96 N    sarves  N     555    806    555
#> 2         82 W    deceits W     657    728    657
#> 3         37 W    nothings N      NA    552    712
#> 4         52 N    chuespies N     427    539    427

```

```

#> 5          96 N      mowny      N      1352    1020    1352
#> 6          96 N      cranned     N       907     573     907
#> 7          89 N       flud       N       742     834     742
#> 8           3 N      bromble     N       523     502     523
#> 9           7 N      trubbles    N       782     458     782
#> 10         35 N      playfound   N       643     663     643
#> # ... with 81 more rows

```

Just as we did with `select`, we can precede the indices with a minus sign to drop the corresponding observations. Thus, for example, we can drop the first 10 observations as follows.

```

slice(blp_df, -(1:10))
#> # A tibble: 990 x 7
#>   participant lex  spell      resp      rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl> <dbl> <dbl>
#> 1         82 W   deceits  W       657     728     657
#> 2         37 W   nothings N        NA     552     712
#> 3         52 N   chuespies N       427     539     427
#> 4         96 N   mowny    N      1352    1020    1352
#> 5         96 N   cranned  N       907     573     907
#> 6         89 N    flud     N       742     834     742
#> 7          3 N   bromble  N       523     502     523
#> 8          7 N   trubbles N       782     458     782
#> 9         35 N   playfound N       643     663     643
#> 10        46 W   mirage   W       778     571     778
#> # ... with 980 more rows

```

A useful `dplyr` function that can be used in `slice` and elsewhere is `n()`, which gives the number of observations in the data frame. Using this, we can, for example, list the observation from index 600 to the end as follows.

```

slice(blp_df, 600:n())
#> # A tibble: 401 x 7
#>   participant lex  spell      resp      rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl> <dbl> <dbl>
#> 1         16 W   earthworms W       767     659     767
#> 2         50 W   markers  W       664     852     664
#> 3         35 N   spoton   N       522     721     522
#> 4         88 W   tawny    N        NA     535     856
#> 5         51 N   gember   N       562     598     562
#> 6         63 W   classed  W       706     429     706
#> 7         63 N   clallers N       401     495     401
#> 8          8 W   pauper   W       734    1126     734
#> 9          2 W   badges   W       485     498     485
#> 10        97 N   foarded  N       802     464     802
#> # ... with 391 more rows

```

Likewise, we could list the last 11 rows as follows.

```

slice(blp_df, (n()-10):n())
#> # A tibble: 11 x 7
#>   participant lex  spell      resp      rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl> <dbl> <dbl>
#> 1         29 N   khandles N       511     777     511
#> 2         88 N   ixcurs  N       504     552     504
#> 3         50 N   homply  N       518     583     518
#> 4        103 W   baste   W       683     454     683
#> 5         67 W   tall    W       476     572     476

```

```

#> 6      45 W      gardens W      586    1023    586
#> 7     105 W     goldfinch N      NA      903    775
#> 8      72 W     varmint N      NA      507    653
#> 9       3 W     lurked W      537    520    537
#> 10      3 W     village W      538    522    538
#> 11     17 W     fudge W      410    437    410

```

The `filter` command is a powerful means to filter observations according to their values. Note that when we say that `filter` filters observations, we mean it filters them *in*, or keeps them, rather than filters them *out*, removes them. For example, we can select all the observations where the `lex` variable is N as follows.

```

filter(blp_df, lex == 'N')
#> # A tibble: 502 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      20 N staud N 977 511 977
#> 2       9 N dinbuss N 565 765 565
#> 3      47 N snilling N 562 496 562
#> 4     103 N gancens N 572 656 572
#> 5      96 N sarves N 555 806 555
#> 6      52 N chuespies N 427 539 427
#> 7      96 N mowny N 1352 1020 1352
#> 8      96 N cranned N 907 573 907
#> 9      89 N flud N 742 834 742
#> 10      3 N bromble N 523 502 523
#> # ... with 492 more rows

```

Notice that here we must use the `==` equality operator. We can also filter by multiple conditions by listing each one with commas between them. For example, the following gives us the observations where `lex` has the value of N and `resp` has the value of W.

```

filter(blp_df, lex == 'N', resp=='W')
#> # A tibble: 35 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      73 N bunding W NA 978 1279
#> 2      63 N gallays W NA 589 923
#> 3      50 N droper W NA 741 573
#> 4       6 N flooder W NA 524 557
#> 5      73 N khantum W NA 623 1355
#> 6      81 N seaped W NA 765 691
#> 7      43 N gafers W NA 556 812
#> 8     101 N winchers W NA 632 852
#> 9      81 N flaged W NA 674 609
#> 10     11 N frocker W NA 653 665
#> # ... with 25 more rows

```

The following gives us those observations where where `lex` has the value of N and `resp` has the value of W and `rt.raw` is less than or equal to 500.

```

filter(blp_df, lex == 'N', resp=='W', rt.raw <= 500)
#> # A tibble: 5 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      28 N cown W NA 680 498
#> 2      17 N beeched W NA 450 469
#> 3      29 N conform W NA 495 497

```

```
#> 4      35 N    blear    W      NA    592    461
#> 5      89 N   stumming W      NA    571    442
```

This command is equivalent to making a conjunction of conditions using `&` as follows.

```
filter(blp_df, lex == 'N' & resp=='W' & rt.raw <= 500)
#> # A tibble: 5 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl>  <dbl>  <dbl>
#> 1      28 N    cown     W      NA    680    498
#> 2      17 N   beeched W      NA    450    469
#> 3      29 N   conform W      NA    495    497
#> 4      35 N    blear    W      NA    592    461
#> 5      89 N   stumming W      NA    571    442
```

We can make a *disjunction* of conditions for filtering using the logical-or symbol `|`. For example, to filter observation where the `rt.raw` was either less than 500 or greater than 1000, we can do the following.

```
filter(blp_df, rt.raw < 500 | rt.raw > 1000)
#> # A tibble: 296 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl>  <dbl>  <dbl>
#> 1      52 N   chuespies N    427    539    427
#> 2      96 N    mowny    N   1352   1020   1352
#> 3      28 W    stelae    N     NA    678    497
#> 4      85 W   forewarned N     NA    525    350
#> 5      24 W     owl    W    470    535    470
#> 6      97 W     soda    W    436    447    436
#> 7      81 N    fugate    N    425    403    425
#> 8     105 N    pamps     N     NA    884   1494
#> 9      27 W   outgrowth N     NA    633   1014
#> 10     82 W     kitty    W    431    476    431
#> # ... with 286 more rows
```

If we want to filter by observations whose values of certain variables are in a set, we can use the `%in%` operator. For example, here we filter observations where values of `rt.raw` is in the set on integers 500 to 510.

```
filter(blp_df, rt.raw %in% 500:510)
#> # A tibble: 26 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl>  <dbl>  <dbl>
#> 1      44 W   subscribed W    509    475    509
#> 2      89 W    snatcher  W    506   1004    506
#> 3       2 N   tronculling N    508    490    508
#> 4      43 N   trabnate  N    510    542    510
#> 5      75 N   dousleens N    508    924    508
#> 6      94 W   strangeness W    508    522    508
#> 7      68 W     greed    W    505    653    505
#> 8      32 N    krifo     N    508    607    508
#> 9       2 W    tweaks    W    508    474    508
#> 10     85 N    waffs     N    506    471    506
#> # ... with 16 more rows
```

In general, we may filter the observations by creating any complex Boolean conditional using combinations of logical-and `&`, logical-or `|`, logical-not `!`, and other operators. For example, here is where the `lex` is `W`, the length of the `spell` is less than 5 and either the `resp` is not equal to `lex` or the `rt.raw` is greater than 900.

```
filter(blp_df,
```

```

      lex == 'W',
      str_length(spell) < 5 & (resp != lex | rt.raw > 900))
#> # A tibble: 14 x 7
#>   participant lex  spell resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      21 W   bosk  N      NA     608  1532
#> 2      68 W   wily  N      NA     723   636
#> 3      30 W   sew   N      NA     473   524
#> 4      34 W   jibs  N      NA     781   756
#> 5      85 W   rote  N      NA     505   458
#> 6      13 W   oofs  N      NA     560   654
#> 7      72 W   awed  N      NA    1203  1801
#> 8      14 W   yids  N      NA     625   620
#> 9      68 W   oho   N      NA     633   630
#> 10     103 W   carl  N      NA    1046  1042
#> 11      46 W   brae  N      NA     644   720
#> 12      81 W   bloc  N      NA     759   575
#> 13      75 W   kind  W      903   1067   903
#> 14      67 W   irk   N      NA     605   570

```

The `filter` command has the variants `filter_all`, `filter_at`, and `filter_if`. In these commands, filtering is applied on the basis of the values of selected sets of variables. For example, using `filter_all`, we can filter rows that contain at least one NA value.

```

filter_all(blpl_df, any_vars(is.na(.)))
#> # A tibble: 179 x 7
#>   participant lex  spell      resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>      <chr> <dbl> <dbl> <dbl>
#> 1      37 W   nothings  N      NA     552   712
#> 2      28 W   stelae    N      NA     678   497
#> 3      85 W   forewarned N      NA     525   350
#> 4     105 N   pamps     N      NA     884  1494
#> 5      27 W   outgrowth N      NA     633  1014
#> 6      89 W   chards    N      NA     545   754
#> 7      63 N   shrudule  N      NA      0  2553
#> 8      73 W   chiggers  N      NA     726   654
#> 9      73 N   bunding   W      NA     978  1279
#> 10     22 W   aitches   N      NA     521   665
#> # ... with 169 more rows

```

In this case, the `.` signifies the variables that are selected, which in the case of `filter_all` is all variables. Thus, this command is filtering observations where any variable contains a NA. On the other hand, to apply the filtering rules to a selected set of variables we can use `filter_at`. For example, the following filters all observations where the value of all variables that start or end with `rt` are greater than 500.

```

filter_at(blpl_df, vars(matches('^rt|rt$')), all_vars(. > 500))
#> # A tibble: 530 x 7
#>   participant lex  spell      resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>      <chr> <dbl> <dbl> <dbl>
#> 1      20 N   staud     N      977     511   977
#> 2       9 N   dinbuss   N      565     765   565
#> 3     103 N   gancens   N      572     656   572
#> 4      45 W   filled    W      659     981   659
#> 5      73 W   journals  W      538    1505   538
#> 6      24 W   apache    W      626     546   626
#> 7      11 W   flake     W      566     717   566

```



```

#> 8      32 W      reliefs W      922  1471  922
#> 9      96 N      sarves  N      555   806  555
#> 10     82 W      deceits W      657   728  657
#> # ... with 520 more rows

```

As another example, the following filters all observations where the value of all variables that start or end with `rt` have values that are less than the median values of those values. In other words, all filtered observations have values of the `rt` variables that are lower than the medians of these variables.

```

filter_at(blp_df,
  vars(matches('^rt|rt$')),
  all_vars(. < median(., na.rm=T)))
#> # A tibble: 251 x 7
#>   participant lex  spell  resp  rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      47 N  snilling N    562  496  562
#> 2      52 N  chuespies N    427  539  427
#> 3       3 N  bromble  N    523  502  523
#> 4      36 W  outposts W    560  461  560
#> 5      24 W  owl    W    470  535  470
#> 6      97 W  soda     W    436  447  436
#> 7      18 N  tesslier N    560  477  560
#> 8      81 N  fugate   N    425  403  425
#> 9      29 N  placker  N    542  558  542
#> 10     82 W  kitty    W    431  476  431
#> # ... with 241 more rows

```

The `filter_if` variant of `filter`, like `select_if` or `rename_if`, allows us to select variables according to their properties, rather than their names, and then apply filtering commands to the selected variables. For example, we can select the numeric variables in the data frames and then filter the observations where all the values of the selected variables are less than the median value of these variables.

```

filter_if(blp_df,
  is.numeric,
  all_vars(. < median(., na.rm=T)))
#> # A tibble: 138 x 7
#>   participant lex  spell  resp  rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1         3 N  bromble N    523  502  523
#> 2        36 W  outposts W    560  461  560
#> 3        24 W  owl    W    470  535  470
#> 4        18 N  tesslier N    560  477  560
#> 5        29 N  placker  N    542  558  542
#> 6         6 N  checsons N    491  555  491
#> 7        19 N  jontage N    413  471  413
#> 8        44 W  snows    W    437  432  437
#> 9        13 N  lavo     N    479  510  479
#> 10       17 N  basyl    N    413  508  413
#> # ... with 128 more rows

```

## Changing variables and values with `mutate`

The `mutate` command is a very powerful tool in the `dplyr` toolbox. It allows us to create new variables and alter the values of existing ones.

As an example, we can create a new variable `is_accurate` that takes the value of `TRUE` whenever `lex` and `resp` have the same value as follows.

```
mutate(blp_df, acc = lex == resp)
#> # A tibble: 1,000 x 8
#>   participant lex  spell  resp  rt prev.rt rt.raw acc
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl> <lgl>
#> 1      20 N   staud  N    977   511   977 TRUE
#> 2       9 N   dinbuss N    565   765   565 TRUE
#> 3      47 N   snilling N    562   496   562 TRUE
#> 4     103 N   gancens N    572   656   572 TRUE
#> 5      45 W   filled  W    659   981   659 TRUE
#> 6      73 W   journals W    538  1505   538 TRUE
#> 7      24 W   apache  W    626   546   626 TRUE
#> 8      11 W   flake   W    566   717   566 TRUE
#> 9      32 W   reliefs W    922  1471   922 TRUE
#> 10     96 N   sarves  N    555   806   555 TRUE
#> # ... with 990 more rows
```

As another example, we can create a new variable that gives the length of the word given by the `spell` variable.

```
mutate(blp_df, len = str_length(spell))
#> # A tibble: 1,000 x 8
#>   participant lex  spell  resp  rt prev.rt rt.raw  len
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl> <int>
#> 1      20 N   staud  N    977   511   977   5
#> 2       9 N   dinbuss N    565   765   565   7
#> 3      47 N   snilling N    562   496   562   8
#> 4     103 N   gancens N    572   656   572   7
#> 5      45 W   filled  W    659   981   659   6
#> 6      73 W   journals W    538  1505   538   8
#> 7      24 W   apache  W    626   546   626   6
#> 8      11 W   flake   W    566   717   566   5
#> 9      32 W   reliefs W    922  1471   922   7
#> 10     96 N   sarves  N    555   806   555   6
#> # ... with 990 more rows
```

We can also create multiple new variable at the same time as in the following example.

```
mutate(blp_df,
  acc = lex == resp,
  fast = rt.raw < mean(rt.raw, na.rm=TRUE))
#> # A tibble: 1,000 x 9
#>   participant lex  spell  resp  rt prev.rt rt.raw acc  fast
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl> <lgl> <lgl>
#> 1      20 N   staud  N    977   511   977 TRUE FALSE
#> 2       9 N   dinbuss N    565   765   565 TRUE  TRUE
#> 3      47 N   snilling N    562   496   562 TRUE  TRUE
#> 4     103 N   gancens N    572   656   572 TRUE  TRUE
#> 5      45 W   filled  W    659   981   659 TRUE  TRUE
#> 6      73 W   journals W    538  1505   538 TRUE  TRUE
#> 7      24 W   apache  W    626   546   626 TRUE  TRUE
#> 8      11 W   flake   W    566   717   566 TRUE  TRUE
#> 9      32 W   reliefs W    922  1471   922 TRUE FALSE
#> 10     96 N   sarves  N    555   806   555 TRUE  TRUE
#> # ... with 990 more rows
```

As with other dplyr verbs, `mutate` has `mutate_all`, `mutate_at`, `mutate_if` variants. The `mutate_all`

variant will apply a transformation function to all variables in the data frame, and then replace the original values of all variables with the results of the function. For example, the following will apply the `as.character` function, which converts any vector into a character vector, to all the variables in `blp_df`.

```
mutate_all(blp_df, as.character)
#> # A tibble: 1,000 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <chr> <chr> <chr> <chr> <chr> <chr> <chr>
#> 1 20 N staud N 977 511 977
#> 2 9 N dinbuss N 565 765 565
#> 3 47 N snilling N 562 496 562
#> 4 103 N gancens N 572 656 572
#> 5 45 W filled W 659 981 659
#> 6 73 W journals W 538 1505 538
#> 7 24 W apache W 626 546 626
#> 8 11 W flake W 566 717 566
#> 9 32 W reliefs W 922 1471 922
#> 10 96 N sarves N 555 806 555
#> # ... with 990 more rows
```

The `mutate_at` variant allows us to apply a function to selected variables. For example, we could apply a log transform to all the `rt` variables as follows.

```
mutate_at(blp_df, vars(matches('^rt|rt$')), log)
#> # A tibble: 1,000 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1 20 N staud N 6.88 6.24 6.88
#> 2 9 N dinbuss N 6.34 6.64 6.34
#> 3 47 N snilling N 6.33 6.21 6.33
#> 4 103 N gancens N 6.35 6.49 6.35
#> 5 45 W filled W 6.49 6.89 6.49
#> 6 73 W journals W 6.29 7.32 6.29
#> 7 24 W apache W 6.44 6.30 6.44
#> 8 11 W flake W 6.34 6.58 6.34
#> 9 32 W reliefs W 6.83 7.29 6.83
#> 10 96 N sarves N 6.32 6.69 6.32
#> # ... with 990 more rows
```

The `mutate_if` variant selects variable by their properties and then applies a function to the selected variables. In the following example, we select all variables that are character vectors and convert them to a *factor*, which is a categorical variable vector with an defined set of values or “levels”, using the `as.factor` function.

```
mutate_if(blp_df, is.character, as.factor)
#> # A tibble: 1,000 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <dbl> <fct> <fct> <fct> <dbl> <dbl> <dbl>
#> 1 20 N staud N 977 511 977
#> 2 9 N dinbuss N 565 765 565
#> 3 47 N snilling N 562 496 562
#> 4 103 N gancens N 572 656 572
#> 5 45 W filled W 659 981 659
#> 6 73 W journals W 538 1505 538
#> 7 24 W apache W 626 546 626
#> 8 11 W flake W 566 717 566
#> 9 32 W reliefs W 922 1471 922
```

```
#> 10      96 N      sarves  N      555      806      555
#> # ... with 990 more rows
```

*Recoding:* We have a number of options to use with `mutate` and its variants for recoding the values of variables. Perhaps the simplest option is `if_else`. This evaluates a condition for each value of a variable. If the result is `TRUE`, it returns one value, other it returns another. As an example, the following code creates a new variable `speed` that takes the value of `fast` if `rt.raw` is less than 750, and takes the value of `slow` otherwise.

```
mutate(blp_df,
      speed = if_else(rt.raw < 750,
                     'fast',
                     'slow')
)
#> # A tibble: 1,000 x 8
#>   participant lex      spell      resp      rt prev.rt rt.raw speed
#>   <dbl> <chr> <chr>      <chr> <dbl> <dbl> <dbl> <chr>
#> 1      20 N      staud      N      977      511      977 slow
#> 2       9 N      dinbuss    N      565      765      565 fast
#> 3      47 N      snilling  N      562      496      562 fast
#> 4     103 N      gancens   N      572      656      572 fast
#> 5      45 W      filled    W      659      981      659 fast
#> 6      73 W      journals  W      538     1505      538 fast
#> 7      24 W      apache    W      626      546      626 fast
#> 8      11 W      flake     W      566      717      566 fast
#> 9      32 W      reliefs   W      922     1471      922 slow
#> 10     96 N      sarves    N      555      806      555 fast
#> # ... with 990 more rows
```

Another widely used recoding method is `recode`. For example, to replace the `lex` variable's values `W` and `N` with `word` and `nonword`, we would do the following.

```
mutate(blp_df,
      lex = recode(lex, 'W'='word', 'N'='nonword')
)
#> # A tibble: 1,000 x 7
#>   participant lex      spell      resp      rt prev.rt rt.raw
#>   <dbl> <chr> <chr>      <chr> <dbl> <dbl> <dbl>
#> 1      20 nonword staud      N      977      511      977
#> 2       9 nonword dinbuss    N      565      765      565
#> 3      47 nonword snilling  N      562      496      562
#> 4     103 nonword gancens   N      572      656      572
#> 5      45 word    filled    W      659      981      659
#> 6      73 word    journals  W      538     1505      538
#> 7      24 word    apache    W      626      546      626
#> 8      11 word    flake     W      566      717      566
#> 9      32 word    reliefs   W      922     1471      922
#> 10     96 nonword sarves    N      555      806      555
#> # ... with 990 more rows
```

Given that both `lex` and `resp` are coded identically, we can apply the same recoding rule to both using `mutate_at` as in the following example.

```
mutate_at(blp_df,
          vars(lex, resp),
          ~recode(., 'W'="word", 'N'="nonword")
)
#> # A tibble: 1,000 x 7
```

```

#>   participant lex      spell      resp      rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      20 nonword staud      nonword  977    511   977
#> 2       9 nonword dinbuss nonword  565    765   565
#> 3      47 nonword snilling nonword  562    496   562
#> 4     103 nonword gancens nonword  572    656   572
#> 5      45 word   filled   word    659    981   659
#> 6      73 word   journals word    538   1505   538
#> 7      24 word   apache   word    626    546   626
#> 8      11 word   flake    word    566    717   566
#> 9      32 word   reliefs  word    922   1471   922
#> 10     96 nonword sarves   nonword  555    806   555
#> # ... with 990 more rows

```

When we are recoding numeric vales using `recode`, we must surround the values we would like to transform using backticks as in the following example.

```

mutate(bl_p_df, rt = recode(rt, `977` = 1000, `562` = 100))
#> # A tibble: 1,000 x 7
#>   participant lex      spell      resp      rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      20 N      staud      N      1000    511   977
#> 2       9 N      dinbuss   N      565    765   565
#> 3      47 N      snilling  N      100    496   562
#> 4     103 N      gancens   N      572    656   572
#> 5      45 W      filled   W      659    981   659
#> 6      73 W      journals  W      538   1505   538
#> 7      24 W      apache   W      626    546   626
#> 8      11 W      flake    W      566    717   566
#> 9      32 W      reliefs  W      922   1471   922
#> 10     96 N      sarves   N      555    806   555
#> # ... with 990 more rows

```

For more complex recoding operations we can use the `case_when` function. For example, we could use `case_when` to convert values of `prev.rt` that are below 500 to `fast`, and those above 1500 to `slow`, and those in between 500 and 1500 to `medium`.

```

mutate(bl_p_df,
  prev.rt = case_when(
    prev.rt < 500 ~ 'fast',
    prev.rt > 1500 ~ 'slow',
    TRUE ~ 'medium'
  )
)
#> # A tibble: 1,000 x 7
#>   participant lex      spell      resp      rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <chr> <dbl>
#> 1      20 N      staud      N      977 medium  977
#> 2       9 N      dinbuss   N      565 medium  565
#> 3      47 N      snilling  N      562 fast    562
#> 4     103 N      gancens   N      572 medium  572
#> 5      45 W      filled   W      659 medium  659
#> 6      73 W      journals  W      538 slow    538
#> 7      24 W      apache   W      626 medium  626
#> 8      11 W      flake    W      566 medium  566
#> 9      32 W      reliefs  W      922 medium  922

```

```
#> 10          96 N    sarves  N          555 medium    555
#> # ... with 990 more rows
```

On each line of `case_when` we have a `~`. To the left of `~`, we have a condition. To the right, we have the replacement value for those values for which the condition is true. Whichever condition first evaluates as true will determine which replacement value is used. For example, in the following example, values lower than 500 are classified as `extra-fast` and values lower than 550 are classified as `fast`. Clearly, any value that is less than 550 is also less than 500, but whichever condition first evaluates to true will determine the replacement value. As such, in the following example, values lower than 500 will be replaced by `extra-fast`.

```
mutate(blp_df,
  prev.rt = case_when(
    prev.rt < 500 ~ 'extra-fast',
    prev.rt < 550 ~ 'fast',
    TRUE ~ 'not-fast'
  )
)
#> # A tibble: 1,000 x 7
#>   participant lex  spell  resp  rt prev.rt  rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <chr> <dbl>
#> 1      20 N    staud  N     977 fast     977
#> 2       9 N    dinbuss N     565 not-fast  565
#> 3      47 N    snilling N     562 extra-fast 562
#> 4     103 N    gancens N     572 not-fast  572
#> 5      45 W    filled  W     659 not-fast  659
#> 6      73 W    journals W     538 not-fast  538
#> 7      24 W    apache  W     626 fast     626
#> 8      11 W    flake   W     566 not-fast  566
#> 9      32 W    reliefs W     922 not-fast  922
#> 10     96 N    sarves  N     555 not-fast  555
#> # ... with 990 more rows
```

On the other hand, in the following example, values lower than 500 will be listed as `fast`, rather than `extra-fast`.

```
mutate(blp_df,
  prev.rt = case_when(
    prev.rt < 550 ~ 'fast',
    prev.rt < 500 ~ 'extra-fast',
    TRUE ~ 'not-fast'
  )
)
#> # A tibble: 1,000 x 7
#>   participant lex  spell  resp  rt prev.rt  rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <chr> <dbl>
#> 1      20 N    staud  N     977 fast     977
#> 2       9 N    dinbuss N     565 not-fast  565
#> 3      47 N    snilling N     562 fast     562
#> 4     103 N    gancens N     572 not-fast  572
#> 5      45 W    filled  W     659 not-fast  659
#> 6      73 W    journals W     538 not-fast  538
#> 7      24 W    apache  W     626 fast     626
#> 8      11 W    flake   W     566 not-fast  566
#> 9      32 W    reliefs W     922 not-fast  922
#> 10     96 N    sarves  N     555 not-fast  555
#> # ... with 990 more rows
```

The final line in the `case_when` above has `TRUE` in place of a condition. This ensures that if any value does not meet any of the previous conditions, it will be assigned the corresponding replacement value in this final line. Had we left this final line out, then any values not meeting the previous conditions would be replaced by `NA`, as seen in the following example.

```
mutate(bl_p_df,
  prev.rt = case_when(
    prev.rt < 550 ~ 'fast',
    prev.rt < 500 ~ 'extra-fast'
  )
)
#> # A tibble: 1,000 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl> <chr>    <dbl>
#> 1      20 N    staud    N     977 fast     977
#> 2       9 N    dinbuss  N     565 <NA>    565
#> 3      47 N    snilling N     562 fast     562
#> 4     103 N    gancens  N     572 <NA>    572
#> 5      45 W    filled  W     659 <NA>    659
#> 6      73 W    journals W     538 <NA>    538
#> 7      24 W    apache  W     626 fast     626
#> 8      11 W    flake   W     566 <NA>    566
#> 9      32 W    reliefs W     922 <NA>    922
#> 10     96 N    sarves  N     555 <NA>    555
#> # ... with 990 more rows
```

Another useful recoding function is `mapvalues`, which is part of the `plyr` package. This allows us to see up two vectors, `from` and `to`, that are of the same length. Any value that matches a value in the `from` is mapped to its corresponding value in `to`. As an example, if we wanted to map the range of integers from 500 to 1000 to the reverse of this range, i.e. 1000, 999, ... 500, we could do the following.

```
mutate(bl_p_df,
  rt_reverse = plyr::mapvalues(rt, from=500:1000, to=1000:500)
)
#> # A tibble: 1,000 x 8
#>   participant lex    spell    resp    rt prev.rt rt.raw rt_reverse
#>   <dbl> <chr> <chr>    <chr> <dbl> <dbl> <dbl> <dbl>
#> 1      20 N    staud    N     977    511    977    523
#> 2       9 N    dinbuss  N     565    765    565    935
#> 3      47 N    snilling N     562    496    562    938
#> 4     103 N    gancens  N     572    656    572    928
#> 5      45 W    filled  W     659    981    659    841
#> 6      73 W    journals W     538   1505    538    962
#> 7      24 W    apache  W     626    546    626    874
#> 8      11 W    flake   W     566    717    566    934
#> 9      32 W    reliefs W     922   1471    922    578
#> 10     96 N    sarves  N     555    806    555    945
#> # ... with 990 more rows
```

*Transmuting:* A variant of `mutate` is `transmute`, which has the `_all`, `_at`, and `_if` variants too. The `transmute` function works like `mutate` except that it only returns the newly created variables, and so drops all the original variables. For example, in the following code, we create two new variables and only these are returned by the `transmute` function.

```
transmute(bl_p_df,
  speed = rt.raw / 1000,
```

```

      accuracy = lex == resp)
#> # A tibble: 1,000 x 2
#>   speed accuracy
#>   <dbl> <lgl>
#> 1 0.977 TRUE
#> 2 0.565 TRUE
#> 3 0.562 TRUE
#> 4 0.572 TRUE
#> 5 0.659 TRUE
#> 6 0.538 TRUE
#> 7 0.626 TRUE
#> 8 0.566 TRUE
#> 9 0.922 TRUE
#> 10 0.555 TRUE
#> # ... with 990 more rows

```

## Sorting observations with arrange

Sorting observations in a data frame is easily accomplished with `arrange`. For example to sort by `participant` and then by `spell`, we would do the following.

```

arrange(blp_df, participant, spell)
#> # A tibble: 1,000 x 7
#>   participant lex  spell  resp  rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1     1 W  abyss  W    629  683  629
#> 2     1 N  baises N    524  574  524
#> 3     1 W  carport W    779  605  779
#> 4     1 N  cellies N    792  652  792
#> 5     1 W  chafing W    601  720  601
#> 6     1 N  dametails N    694  635  694
#> 7     1 N  footer N    789  566  789
#> 8     1 W  gantries W    644  581  644
#> 9     1 N  hogtush N    679  568  679
#> 10    1 N  lisedess N    679  619  679
#> # ... with 990 more rows

```

We can sort by the reverse order of any variable by using the `desc` command on the variable. In the following example, we sort by `participant`, and then by `spell` in reverse order.

```

arrange(blp_df, participant, desc(spell))
#> # A tibble: 1,000 x 7
#>   participant lex  spell  resp  rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1     1 N  wintes  N    545  629  545
#> 2     1 N  treeps  N    607  610  607
#> 3     1 W  squashes W    494  491  494
#> 4     1 N  sinkhicks N    536  519  536
#> 5     1 W  shafting W    553  571  553
#> 6     1 W  month  W    500  498  500
#> 7     1 N  lisedess N    679  619  679
#> 8     1 N  hogtush N    679  568  679
#> 9     1 W  gantries W    644  581  644
#> 10    1 N  footer N    789  566  789
#> # ... with 990 more rows

```



## Subsampling data frames

The `dplyr` package provides two methods to sample from a data frame. The `sample_frac` allows us to sample a specified proportion of observations. In the following example, we randomly sample 10% of the data frame.

```
sample_frac(blp_df, 0.1)
#> # A tibble: 100 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl> <dbl> <dbl>
#> 1     17 W    beets    N      NA    672  1115
#> 2    103 N    wisked   N      NA    701  1159
#> 3     10 W    clung    W      NA   1835  9925
#> 4     43 W    huddled  W      717   557   717
#> 5     72 W    shocks  W      586   554   586
#> 6     94 N    hatchpock N     559   726   559
#> 7     14 N    nagur    N     539   639   539
#> 8      3 W    monday  W     411   705   411
#> 9     36 W    revived  W     664   652   664
#> 10    105 N    fondism  N     827   541   827
#> # ... with 90 more rows
```

By default, the sampling will occur without replacement, which we can override as follows.

```
sample_frac(blp_df, 0.1, replace=FALSE)
#> # A tibble: 100 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl> <dbl> <dbl>
#> 1     17 W    hauteur  N      NA    479   789
#> 2     67 W    advanced W     636   641   636
#> 3     52 N    nerncades N     443   856   443
#> 4     77 W    needed   W     512   394   512
#> 5     79 W    wreath   W     495   593   495
#> 6     64 W    twined   W     795   474   795
#> 7     10 W    prolix   N      NA    439   380
#> 8     54 N    tincers  N     623  1027   623
#> 9     68 N    boaves   N     618   470   618
#> 10    97 W    petalled W     798   676   798
#> # ... with 90 more rows
```

We may also sample a specified number of observations, as in the following example, where we randomly sample 15 observations.

```
sample_n(blp_df, 15)
#> # A tibble: 15 x 7
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl> <dbl> <dbl>
#> 1     96 W    service  W     614   617   614
#> 2     97 W    remits   W     544   577   544
#> 3     34 W    sources  W     479   460   479
#> 4    103 W    apace    N      NA    529   514
#> 5      8 W    sketched W     549   884   549
#> 6     32 W    carboys  N      NA     39   460
#> 7     97 W    soda     W     436   447   436
#> 8     86 W    only     W     412   586   412
#> 9     32 W    cheeped  N      NA    471  1099
#> 10    75 N    perklines N    1750  1184  1750
```

```

#> 11      18 N      moorwack N      460      456      460
#> 12      37 W      gospel   W      596      665      596
#> 13      63 N      gallays  W      NA       589      923
#> 14      77 W      yobs     W      522     1054      522
#> 15      42 W      icepack  W      574      476      574

```

We may also sample the top or bottom observations according to some variable. For example, here we select the top 15 observations by their `rt.raw` values.

```

top_n(blp_df, 15, rt.raw)
#> # A tibble: 15 x 7
#>   participant lex spell resp rt prev.rt rt.raw
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      63 N shrudule N NA 0 2553
#> 2      51 W trumping W NA 670 2777
#> 3      73 W plank N NA 631 1939
#> 4      65 W savers N NA 1168 5815
#> 5      70 N ashdess N NA 510 2256
#> 6      68 N howned N NA 522 2891
#> 7      85 W twitted W NA 1029 2625
#> 8      65 W forenames W NA 471 4537
#> 9      78 N gassolled N NA 755 2362
#> 10     12 W coursed N NA 1054 3434
#> 11     54 W puffer N NA 582 1972
#> 12    105 N fragrents N NA 1090 2554
#> 13     10 W clung W NA 1835 9925
#> 14     90 N clate N NA 1051 2199
#> 15     66 W submersed W NA 2199 3029

```

## Reducing data with `summarize` and `group_by`

The `dplyr` package has a function `summarize` (or, equivalently, `summarise`) that applies summarizing functions to variables. A summarizing function is essentially any function that takes a vector and reduces it to a single values. The `summarize` function is vital for exploratory data analysis and we will use it extensively in Chapter 5. However, for now, especially when used with the `group_by` function, it is an essential tool for data wrangling.

To see how `summarize` works, we may calculate some summary statistics of the particular variables as in the following example.

```

summarize(blp_df,
  mean_rt = mean(rt, na.rm = T),
  median_rt = median(rt, na.rm = T),
  sd_rt.raw = sd(rt.raw, na.rm = T)
)
#> # A tibble: 1 x 3
#>   mean_rt median_rt sd_rt.raw
#>   <dbl> <dbl> <dbl>
#> 1 638. 588 474.

```

(Note that here it is necessary to use `na.rm = T` to remove the NA values in the variables.)

We can use the `summarize_all` variant of `summarize` to apply a summarisation function to all variables, as in the following example.

```

summarize_all(blp_df, n_distinct)
#> # A tibble: 1 x 7
#>   participant lex spell resp rt prev.rt rt.raw

```

```
#>           <int> <int> <int> <int> <int>   <int> <int>
#> 1           78     2  990     2  421     493  516
```

Here, `n_distinct` returns the number of unique values in each variable. The `summarize_at` will apply a summary function to selected variables. In the following example, we calculate the mean of all the reaction times variables.

```
summarize_at(blp_df, vars(matches('^rt|rt$')), ~mean(., na.rm=T))
#> # A tibble: 1 x 3
#>   rt prev.rt rt.raw
#>   <dbl> <dbl> <dbl>
#> 1  638.   660.  708.
```

The `summarize_if` will apply the summary function to variables selected by their properties, such as whether they are numeric variables, as in the following example.

```
summarize_if(blp_df, is.numeric, ~mean(., na.rm=T))
#> # A tibble: 1 x 4
#>   participant  rt prev.rt rt.raw
#>   <dbl> <dbl> <dbl> <dbl>
#> 1    49.5  638.   660.  708.
```

Using the `_all`, `_at`, `_if` variants, we can also apply multiple summary functions simultaneously. In the following example, we calculate three summary statistics for `rt` alone.

```
summarise_at(blp_df,
  vars(rt),
  list(mean = ~mean(., na.rm=T),
        median = ~median(., na.rm=T),
        sd = ~sd(., na.rm=T)
  )
)
#> # A tibble: 1 x 3
#>   mean median  sd
#>   <dbl> <dbl> <dbl>
#> 1  638.   588  191.
```

In the following, we calculate the same three summary statistics for two variables.

```
summarise_at(blp_df,
  vars(rt, rt.raw),
  list(mean = ~mean(., na.rm=T),
        median = ~median(., na.rm=T),
        sd = ~sd(., na.rm=T)
  )
)
#> # A tibble: 1 x 6
#>   rt_mean rt.raw_mean rt_median rt.raw_median rt_sd rt.raw_sd
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  638.   708.   588   605  191.   474.
```

In this case, the name of the summary value is appended to the name of each variable.

The `summarize` command, and its variants, become considerably more powerful when combined with the `group_by` command. Effectively, `group_by` groups the observations within a data frame according to the values of specified variables. For example, the following command groups `blp_df` into groups of observations according to value of the `lex` variable.

```
blp_by_lex <- group_by(blp_df, lex)
```

If we view the resulting grouped data frame, it appears more or less as normal.

```
blp_by_lex
#> # A tibble: 1,000 x 7
#> # Groups:   lex [2]
#>   participant lex    spell    resp    rt prev.rt rt.raw
#>   <dbl> <chr> <chr>    <chr> <dbl> <dbl> <dbl>
#> 1      20 N    staud    N     977    511    977
#> 2       9 N    dinbuss  N     565    765    565
#> 3      47 N    snilling N     562    496    562
#> 4     103 N    gancens  N     572    656    572
#> 5      45 W    filled  W     659    981    659
#> 6      73 W    journals W     538   1505    538
#> 7      24 W    apache  W     626    546    626
#> 8      11 W    flake   W     566    717    566
#> 9      32 W    reliefs W     922   1471    922
#> 10     96 N    sarves  N     555    806    555
#> # ... with 990 more rows
```

Like `blp_df`, it has 1000 observations and 7 variables. However, in addition, it is comprised of 2 groups that are defined by the values of the `lex` variable.

If we now apply `summarize` to this grouped data frame, we will obtain summary statistics for each group, as in the following example.

```
summarize(blp_by_lex, mean = mean(rt, na.rm=T))
#> # A tibble: 2 x 2
#>   lex    mean
#>   <chr> <dbl>
#> 1 N     638.
#> 2 W     637.
```

We may also apply the `_all`, `_at`, `_if` variants as before.

```
summarize_at(blp_by_lex,
             vars(rt),
             list(mean = ~mean(., na.rm=T),
                  median = ~median(., na.rm=T),
                  sd = ~sd(., na.rm=T)
             )
)
#> # A tibble: 2 x 4
#>   lex    mean median    sd
#>   <chr> <dbl> <dbl> <dbl>
#> 1 N     638.   585  198.
#> 2 W     637.   588  183.
```

Using `group_by` and `summarize` together is a powerful way to create new (reduced) data frames. For example, in `blp_df`, there are 78 unique participants. For each participant, and for each of the two stimuli types (i.e. the N and W values of `lex`), we can calculate the number of stimuli they were shown (using the `dplyr` command `n()`, which calculates the number of observations per each group), their number of accurate responses and their average response reaction time.

```
summarize(group_by(blp_df, participant, lex),
          n_stimuli = n(),
          correct_resp = sum(resp == lex, na.rm=T),
          reaction_time = mean(rt.raw, na.rm=T))
#> # A tibble: 156 x 5
```

```

#> # Groups:   participant [78]
#>   participant lex  n_stimuli correct_resp reaction_time
#>     <dbl> <chr>    <int>      <int>      <dbl>
#> 1         1 N         9          9        649.
#> 2         1 W         7          7        600
#> 3         2 N         7          6        625.
#> 4         2 W         6          5        477.
#> 5         3 N         4          4        540.
#> 6         3 W         8          7        529
#> 7         4 N         5          5        589.
#> 8         4 W         5          4        465.
#> 9         5 N         1          1        495
#> 10        5 W         3          2        571
#> # ... with 146 more rows

```

The data frame thus produced has 156 observation: two per each of the 78 participants.

Finally, any grouped data framed can be ungrouped by the `ungroup` command, as in the following example.

```

ungroup(bl_p_by_lex)
#> # A tibble: 1,000 x 7
#>   participant lex  spell  resp  rt prev.rt rt.raw
#>     <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl>
#> 1      20 N  staud  N    977  511  977
#> 2       9 N  dinbuss N    565  765  565
#> 3      47 N  snilling N    562  496  562
#> 4     103 N  gancens N    572  656  572
#> 5      45 W  filled W    659  981  659
#> 6      73 W  journals W    538 1505  538
#> 7      24 W  apache W    626  546  626
#> 8      11 W  flake W    566  717  566
#> 9      32 W  reliefs W    922 1471  922
#> 10     96 N  sarves N    555  806  555
#> # ... with 990 more rows

```

## The %>% operator

The %>% operator in R is known as the *pipe*. It was introduced relatively recently to R, and is a simple yet major innovation. It allows us to create sequences of functions, sometimes known as *pipelines*, that avoid the use of repeated function nested or temporary data structures. The result is usually very clean, readable, and uncluttered code.

The %>% pipe, and related operators like %<>% and %\$% are part of the `magrittr` package. The pipe itself is, however, automatically loaded by the `dplyr` package, as well as by `tidyverse`. In RStudio, the keyboard shortcut Ctrl+Shift+M types %>%.

To understand pipes, let us begin with a very simple example. The following `primes` variable is a vector of the first 10 prime numbers.

```
primes <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

We can calculate the sum of `primes` as follows.

```
sum(primes)
#> [1] 129
```

We may then calculate the square root of this sum.

```
sqrt(sum(primes))
#> [1] 11.35782
```

We may then calculate the logarithm of this square root.

```
log(sqrt(sum(primes)))
#> [1] 2.429906
```

The final calculation is triple nested function. In this example, it is not particularly difficult to read, but often when there is excessive nesting, the result appears cluttered and unreadable. Consider the following example where we combine `primes` with a vector of 3 NA values, subsample 5 values with replacement, sum the result, removing missing values, then calculate the square root, and its logarithm to base 2.

```
log(sqrt(sum(sample(c(primes, rep(NA, 3)), size=5, replace=T), na.rm=T)), base=2)
#> [1] 2.696159
```

We may try to improve the readability of this code by breaking the function over multiple lines.

```
log(
  sqrt(
    sum(
      sample(
        c(primes, rep(NA, 3)),
        size=5,
        replace=T),
      na.rm=T)),
  base=2)
#> [1] 2.261781
```

It is questionable whether this improves readability at all. An alternative approach to improve readability is to create intermediate variables as in the following code.

```
primes_appended <- c(primes, rep(NA, 3))
primes_subsample <- sample(primes_appended, size=5, replace=T)
primes_subsample_sum <- sum(primes_subsample, na.rm=T)
sqrt_primes_subsample_sum <- sqrt(primes_subsample_sum)
log(sqrt_primes_subsample_sum, base=2)
#> [1] 3.022197
```

Or, alternatively, we could re-use the same temporary variable for the intermediate calculations.

```
tmpvar <- c(primes, rep(NA, 3))
tmpvar <- sample(tmpvar, size=5, replace=T)
tmpvar <- sum(tmpvar, na.rm=T)
tmpvar <- sqrt(tmpvar)
log(tmpvar, base=2)
#> [1] 1.403677
```

In either case, the resulting code is relatively cluttered, and creates some unnecessary temporary variables.

The `%>%` is *syntactic sugar* that reexpresses nested functions as sequences. It is binary operator that takes the value of its left hand side and places it inside the function on the right hand side. This is best understood by example. If we have a variable `x` and a function `f()`, we can apply the function to the variable with `f(x)`. This is equivalent to the following.

```
x %>% f() # equivalent to f(x)
```

If, on the other hand, the nested application of a set of functions `f()`, `g()`, and `h()` would be equivalent to the following.

```
x %>% f() %>% g() %>% h() # equivalent to h(g(f(x)))
```

Returning to some of our examples above, we will see how they can be rewritten with pipes. In each case, we will precede the piped version with a comment showing its original version.

```
# sum(primes)
primes %>% sum()
#> [1] 129

# sum(primes, na.rm=T)
primes %>% sum(na.rm=T)
#> [1] 129

# log(sqrt(sum(primes)))
primes %>% sum() %>% sqrt() %>% log()
#> [1] 2.429906

# log(sqrt(sum(primes, na.rm=T)), base=2)
primes %>%
  sum(na.rm=T) %>%
  sqrt() %>%
  log(base=2)
#> [1] 3.505614

# log(sqrt(sum(sample(c(primes, rep(NA, 3)), size=5, replace=T), na.rm=T)), base=2)
primes %>%
  c(rep(NA, 3)) %>%
  sample(size=5, replace=T) %>%
  sum(na.rm=T) %>%
  sqrt() %>%
  log(base=2)
#> [1] 3.104727
```

In each case, we can the pipeline as beginning with some variable or expression, sending that to a function, the output of which is sent as input to the next function in the pipeline, and so on.

When used with the `dplyr` wrangling tools, as well as other tools that we will meet momentarily, we now have a veritable mini-language for data wrangling. For example, in the following code, create some new variables, select, rename, and reorder, some of the variables, and sort by `participant` and then by `speed`.

```
blp_df %>%
  mutate(accuracy = resp == lex,
         stimulus = recode(lex, 'W'='word', 'N'='nonword')
  ) %>%
  select(participant, stimulus, item=spell, accuracy, speed=rt.raw) %>%
  arrange(participant, speed)
#> # A tibble: 1,000 x 5
#>   participant stimulus item      accuracy speed
#>   <dbl> <chr> <chr> <lgl> <dbl>
#> 1         1 word    squashes TRUE    494
#> 2         1 word    month   TRUE    500
#> 3         1 nonword baises TRUE    524
#> 4         1 nonword sinkhicks TRUE    536
#> 5         1 nonword wintes  TRUE    545
#> 6         1 word    shafting TRUE    553
#> 7         1 word    chafing TRUE    601
#> 8         1 nonword treeps  TRUE    607
#> 9         1 word    abyss   TRUE    629
#> 10        1 word    gantries TRUE    644
#> # ... with 990 more rows
```

As another example, in the following code, we filter the data frame by keeping only observations where `lex` takes the value of `W`, then we calculate the word length and the accuracy of the response, rename the `rt.raw` variable, group by word length, calculate the average accuracy and reaction time, select some key variables and sort the result.

```
blp_df %>%
  filter(lex == 'W') %>%
  mutate(word_length = str_length(spell),
         accuracy = resp == lex) %>%
  rename(speed = rt.raw) %>%
  group_by(word_length) %>%
  summarize_at(vars(accuracy, speed), ~mean(., na.rm=T)) %>%
  ungroup() %>%
  select(word_length, accuracy, speed) %>%
  arrange(word_length, accuracy, speed)
#> # A tibble: 9 x 3
#>   word_length accuracy speed
#>   <int>      <dbl> <dbl>
#> 1         3      0.7   551.
#> 2         4     0.744  649.
#> 3         5     0.718  825.
#> 4         6     0.807  723.
#> 5         7     0.821  704.
#> 6         8     0.835  678.
#> 7         9     0.595  914.
#> 8        10     0.714  670.
#> 9        11     0.5   700.
```

## Combining data frames

There are at least three major ways to combine data frames. They are what we'll call *binds*, *joins*, and *set operations*

### Combining data frames with binds

A *bind* operation is a simple operation that either vertically stack data frames that share common variables, or horizontally stack data frames that have the same number of observations.

To illustrate, we will create three small data frames. Here, we use `tibble` to create the data frame. This is very similar to using `data.frame` to create a data frame, like we saw in Chapter 2, but will create a tibble flavoured data frame, which is the common type of data frame in the tidyverse.

```
Df_1 <- tibble(x = c(1, 2, 3),
              y = c(2, 7, 1),
              z = c(0, 2, 7))

Df_2 <- tibble(y = c(5, 7),
              z = c(6, 7),
              x = c(1, 2))

Df_3 <- tibble(a = c(5, 6, 1),
              b = c('a', 'b', 'c'),
              c = c(T, T, F))
```

The `Df_1` and `Df_2` data frames share common variable names. They can be vertically stacked using a `bind_rows` operation.



```

bind_rows(Df_1, Df_2)
#> # A tibble: 5 x 3
#>   x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     0
#> 2     2     7     2
#> 3     3     1     7
#> 4     1     5     6
#> 5     2     7     7

```

Note that the variables, which are in different orders in the two data frames, are aligned properly when bound together. Any number of compatible data frames can be combined using `bind_rows`, as in the following example.

```

bind_rows(Df_1, Df_2, Df_2, Df_1)
#> # A tibble: 10 x 3
#>   x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     0
#> 2     2     7     2
#> 3     3     1     7
#> 4     1     5     6
#> 5     2     7     7
#> 6     1     5     6
#> 7     2     7     7
#> 8     1     2     0
#> 9     2     7     2
#> 10    3     1     7

```

The `Df_1` and `Df_3` data frames have the same number of observations and so can be stacked side by side with a `bind_cols` operation.

```

bind_cols(Df_1, Df_3)
#> # A tibble: 3 x 6
#>   x     y     z     a b     c
#>   <dbl> <dbl> <dbl> <dbl> <chr> <lgl>
#> 1     1     2     0     5 a    TRUE
#> 2     2     7     2     6 b    TRUE
#> 3     3     1     7     1 c    FALSE

```

As with `bind_rows`, `bind_cols` will bind any number of compatible data frames.

```

bind_cols(Df_1, Df_3, Df_3, Df_1)
#> # A tibble: 3 x 12
#>   x...1 y...2 z...3 a...4 b...5 c...6 a...7 b...8 c...9 x...10 y...11 z...12
#>   <dbl> <dbl> <dbl> <dbl> <chr> <lgl> <dbl> <chr> <lgl> <dbl> <dbl> <dbl>
#> 1     1     2     0     5 a    TRUE     5 a    TRUE     1     2     0
#> 2     2     7     2     6 b    TRUE     6 b    TRUE     2     7     2
#> 3     3     1     7     1 c    FALSE    1 c    FALSE    3     1     7

```

In this case, however, as would be the case if the data frames being bound by `bind_cols`, the variable names are appended with digits to make them unique.

## Combining data frames by joins

A *join* operation is a common operation in relational databases using SQL. It allows us to join separate tables according to shared keys. As an example of a join operation on data frames using `dplyr`, consider the `blp_df` data frame. It has a variable `spell` that gives the identity of the stimulus shown on each trial of the

lexical decision experiment. In a separate file, `blp-stimuli.csv` file, we have three additional variables for these stimuli.

```
stimuli <- read_csv('data/blp_stimuli.csv')
stimuli
#> # A tibble: 55,865 x 4
#>   spell    old20    bnc subtlex
#>   <chr>   <dbl> <dbl>   <dbl>
#> 1 a/c      1.95    14      0
#> 2 aas      1.55     9      1
#> 3 aback    1.85   327     15
#> 4 abaft    2         8      2
#> 5 aband    1.95     0      0
#> 6 abase    1.7       6      2
#> 7 abased   1.75     6      0
#> 8 abashed  1.85    57      0
#> 9 abate    1.75    69      5
#> 10 abates  1.75     9      2
#> # ... with 55,855 more rows
```

As can be seen, there are four variables in `stimuli`, the `spell` variable that denotes the stimulus string and three others, i.e. `old20`, `bnc`, and `subtlex`, that describe properties of that stimulus string.

We can join these two data frames with `inner_join`. An `inner_join` operation, like all the `_join` operations we consider here, always operates on two data frames, which we will refer to as the left and right data frames. It searches through the values of variables that are shared by the two data frames in order to find matching values. In `blp_df` and `stimuli`, there is just one shared variable, namely `spell`. Thus, an `inner_join` of `blp_df` and `stimuli` will find values of `spell` on the left hand data frame that occur as values of `spell` on the right hand side. It will then join the corresponding observations of both data frames.

```
inner_join(blp_df, stimuli)
#> # A tibble: 1,000 x 10
#>   participant lex  spell  resp  rt prev.rt rt.raw old20  bnc subtlex
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1      20 N   staud  N    977   511   977  1.85    0     0
#> 2       9 N   dinbuss N    565   765   565  2.9     0     0
#> 3      47 N   snilling N    562   496   562  1.8     0     0
#> 4     103 N   gancens N    572   656   572  2.3     0     0
#> 5      45 W   filled  W    659   981   659  1.45  5340  1336
#> 6      73 W   journals W    538  1505   538  2.7   1030   83
#> 7      24 W   apache  W    626   546   626  2.45   130   17
#> 8      11 W   flake   W    566   717   566  1.5    274   84
#> 9      32 W   reliefs W    922  1471   922  2.25  185    1
#> 10     96 N   sarves  N    555   806   555  1.65    0     0
#> # ... with 990 more rows
```

In general, in an `inner_join`, if the left hand data frame has no values on the shared variables that match those on the right hand data frame, the observations from the left hand data frame are dropped. In addition, all observations on the right hand data frame that do not have matching observations on the left always get dropped too.

In the example above, all observations of `blp_df` had values of `spell` that matched values of the `spell` in `stimuli`. However, consider the following two data frames.

```
Df_a <- tibble(x = c(1, 2, 3),
               y = c('a', 'b', 'c'))
Df_b <- tibble(x = c(2, 3, 4),
```

```
z = c('d', 'e', 'f')
```

In this case, the first value of `x` in `Df_a` does not match any value of `x` in `Df_b`, and so the corresponding observation is dropped in an `inner_join`.

```
inner_join(Df_a, Df_b)
#> # A tibble: 2 x 3
#>   x y z
#>   <dbl> <chr> <chr>
#> 1 2 b d
#> 2 3 c e
```

A `left_join`, on the other hand, will preserve all values on the left and put `NA` as the corresponding values of the right's variables if there are no matching values.

```
left_join(Df_a, Df_b)
#> # A tibble: 3 x 3
#>   x y z
#>   <dbl> <chr> <chr>
#> 1 1 a <NA>
#> 2 2 b d
#> 3 3 c e
```

A `right_join` preserves all observations from the right, and places `NA` as the corresponding values of variables from the left that are not matched.

```
right_join(Df_a, Df_b)
#> # A tibble: 3 x 3
#>   x y z
#>   <dbl> <chr> <chr>
#> 1 2 b d
#> 2 3 c e
#> 3 4 <NA> f
```

With `blp_df` and `stimuli`, because all observations of `spell` in `blp_df` match values of `spell` in `stimuli`, the `inner_join` and `left_join` are identical, which we can verify as follows (using `all_equal`).

```
all_equal(inner_join(blp_df, stimuli),
           left_join(blp_df, stimuli))
#> [1] TRUE
```

On the other hand, there many values of `spell` in `stimuli` that do not match any values of `spell` in `blp_df`. As such, a `right_join` leads to a large number of observations with `NA` values.

```
right_join(blp_df, stimuli)
#> # A tibble: 55,875 x 10
#>   participant lex spell resp rt prev.rt rt.raw old20 bnc subtex
#>   <dbl> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 20 N staud N 977 511 977 1.85 0 0
#> 2 9 N dinbuss N 565 765 565 2.9 0 0
#> 3 47 N snilling N 562 496 562 1.8 0 0
#> 4 103 N gancens N 572 656 572 2.3 0 0
#> 5 45 W filled W 659 981 659 1.45 5340 1336
#> 6 73 W journals W 538 1505 538 2.7 1030 83
#> 7 24 W apache W 626 546 626 2.45 130 17
#> 8 11 W flake W 566 717 566 1.5 274 84
#> 9 32 W reliefs W 922 1471 922 2.25 185 1
```

```
#> 10          96 N      sarves  N      555      806      555  1.65      0      0
#> # ... with 55,865 more rows
```

A `full_join` keeps all observation in both the left and right data frames. If used with `blp_df` and `stimuli`, the result is identical to a `right_join`, as we can verify as follows.

```
all_equal(full_join(blp_df, stimuli),
          right_join(blp_df, stimuli)
)
#> [1] TRUE
```

For the case of `Df_a` and `Df_b`, where observations in both the left and right data frames do not have matches, a `full_join` is as follows.

```
full_join(Df_a, Df_b)
#> # A tibble: 4 x 3
#>   x y     z
#>   <dbl> <chr> <chr>
#> 1     1  a   <NA>
#> 2     2  b     d
#> 3     3  c     e
#> 4     4 <NA>  f
```

In all of the above examples, the data frames shared only one common variable. Consider the following cases.

```
Df_4 <- tibble(x = c(1, 2, 3),
              y = c(2, 7, 1),
              z = c(0, 2, 7))
```

```
Df_5 <- tibble(a = c(1, 1, 7),
              b = c(2, 3, 7),
              c = c('a', 'b', 'c'))
```

The `Df_4` and `Df_5` do not share any common variables. In this case, we need to specify pairs of variables to match on. We have multiple options for how to do this. For example, in the following example, we look for matches between `x` on the left and `a` on the right.

```
inner_join(Df_4, Df_5, by=c('x' = 'a'))
#> # A tibble: 2 x 5
#>   x     y     z     b c
#>   <dbl> <dbl> <dbl> <dbl> <chr>
#> 1     1     2     0     2 a
#> 2     1     2     0     3 b
```

On the other hand, in the following example, we look for matches between `x` and `y` on the left and `a` and `b` on the right.

```
inner_join(Df_4, Df_5, by=c('x' = 'a', 'y' = 'b'))
#> # A tibble: 1 x 4
#>   x     y     z c
#>   <dbl> <dbl> <dbl> <chr>
#> 1     1     2     0 a
```

## Combining data frames by set operations

In `dplyr`, the functions `intersect`, `union`, etc., allow us to combine data frames *that have identical variables* using set operations.

Consider the following data frames.

```
Df_6 <- tibble(x = c(1, 2, 3),
              y = c(4, 5, 6),
              z = c(7, 8, 9))
```

```
Df_7 <- tibble(y = c(6, 7),
              z = c(9, 10),
              x = c(3, 4))
```

Both data frames have the same variables and happen to share a row of observations, even if the variables are in different orders. As such, their intersection and union are as follows.

```
intersect(Df_6, Df_7)
#> # A tibble: 1 x 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     3     6     9
union(Df_6, Df_7)
#> # A tibble: 4 x 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     4     7
#> 2     2     5     8
#> 3     3     6     9
#> 4     4     7    10
```

We may also calculate the set differences between Df\_6 and Df\_7.

```
setdiff(Df_6, Df_7) # Rows in Df_6 not in Df_7
#> # A tibble: 2 x 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     4     7
#> 2     2     5     8
setdiff(Df_7, Df_6) # Rows in Df_7 not in Df_6
#> # A tibble: 1 x 3
#>       y     z     x
#>   <dbl> <dbl> <dbl>
#> 1     7    10     4
```

## Reshaping with `pivot_longer` and `pivot_wider`

A so-called *tidy* data set, at least according to its widespread usage in the context of data analysis using R, is a data set where all rows are observations, all columns are variables, and each variable is a single value. Although what exactly counts as an observation may in fact vary from situation to situation, usually whether a data set is *tidy* or not is quite clear immediately. For example, consider the following data frame.

```
recall_df <- read_csv('data/repeated_measured_a.csv')
recall_df
#> # A tibble: 5 x 4
#>   Subject Neg Neu Pos
#>   <chr>   <dbl> <dbl> <dbl>
#> 1 Faye     26    12    42
#> 2 Jason     29     8    35
#> 3 Jim      32    15    45
```

```
#> 4 Ron      22   10   38
#> 5 Victor   30   13   40
```

In this data frame, for each subject, we have three values, which are their scores on a memory test in three different conditions of an experiment. The conditions are `Neg` (negative), `Neu` (neutral), `Pos` (positive). Arguably, we could describe each row as an observation, namely the observation of all memory scores from a particular subject. However, each column is not a variable. The `Neg`, `Neu`, `Pos` are, in fact, *values* of a variable, namely the condition of the experiment. Therefore, to tidy this data frame, we need a variable for the subject, another for the experiment's condition, and another for the memory score for the corresponding subject in the corresponding condition. To do so, we perform what is sometimes known as a *wide to long* transformation. The `tidyr` package has a function `pivot_longer` for this transformation.

To use `pivot_longer`, we must specify the variables (using the `cols` argument) that we want to pivot from wide to long. In our case, it is the variables `Neg`, `Neu`, `Pos`, and we can select these by `cols = -Subject`, which means all variables except `Subject`. Next, using the argument `names_to`, we must provide a name for the column that will indicate the experimental condition. We will do this with `names_to = 'condition'`. The values of this `condition` variable will consist of the values `Neg`, `Neu`, `Pos`. Finally, using the argument `values_to`, we must provide a name for the column that will indicate the memory scores. We will do this with `values_to = 'score'`. The values of this `score` variable will consist of the values of the original `Neg`, `Neu`, `Pos` columns. Altogether, we have the following.

```
recall_long <- pivot_longer(recall_df,
                             cols = -Subject,
                             names_to = 'condition',
                             values_to = 'score')
```

```
recall_long
#> # A tibble: 15 x 3
#>   Subject condition score
#>   <chr>    <chr>    <dbl>
#> 1 Faye     Neg         26
#> 2 Faye     Neu         12
#> 3 Faye     Pos         42
#> 4 Jason    Neg         29
#> 5 Jason    Neu          8
#> 6 Jason    Pos         35
#> 7 Jim      Neg         32
#> 8 Jim      Neu         15
#> 9 Jim      Pos         45
#> 10 Ron     Neg         22
#> 11 Ron     Neu         10
#> 12 Ron     Pos         38
#> 13 Victor  Neg         30
#> 14 Victor  Neu         13
#> 15 Victor  Pos         40
```

Now, each row is an observation, namely providing the memory score for the given subject in the given condition, and each column is a variable.

Once the data frame is in this format, other operations, such as those using the `dplyr` functions, become much easier. For example, to calculate some summary statistics on the `mem_score` per condition, we would do the following.

```
recall_long %>%
  group_by(condition) %>%
  summarize_at('score', list(median=median,
                              mean=mean,
                              min=min,
```

```

max=max)
)
#> # A tibble: 3 x 5
#>   condition median mean  min  max
#>   <chr>         <dbl> <dbl> <dbl> <dbl>
#> 1 Neg          29  27.8  22   32
#> 2 Neu          12  11.6   8   15
#> 3 Pos          40  40     35  45

```

The inverse of a `pivot_longer` is a `pivot_wider`. It is very similar to `pivot_longer` and we use `names_from` and `values_from` in the opposite sense to `names_to` and `values_to`.

```

pivot_wider(recall_long, names_from = 'condition', values_from = 'score')
#> # A tibble: 5 x 4
#>   Subject Neg  Neu  Pos
#>   <chr>   <dbl> <dbl> <dbl>
#> 1 Faye    26   12  42
#> 2 Jason   29    8  35
#> 3 Jim     32   15  45
#> 4 Ron     22   10  38
#> 5 Victor  30   13  40

```

Some `gather` operations are not as simple as the one just described. Consider the following data.

```

recall_2_df <- read_csv('data/repeated_measured_b.csv')
recall_2_df
#> # A tibble: 5 x 7
#>   Subject Cued_Neg Cued_Neu Cued_Pos Free_Neg Free_Neu Free_Pos
#>   <chr>     <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1 Faye      15     16     14     13     13     12
#> 2 Jason      4      9     10      6      7      9
#> 3 Jim        7      9     10      8      9      5
#> 4 Ron       17     18     20     12     14     15
#> 5 Victor    16     13     14     12     13     14

```

In this data frame, we have 6 columns that are the values of a combination of two experimental variables. One variable is a binary variable that indicates if the experimental condition was **Cued** or **Free** (i.e., was the subject's memory recall cued by some stimuli or was it a free recall). The other variable is the condition as in the `recall_df` data frame. If we perform a `pivot_longer` as we did before we obtain the following.

```

pivot_longer(recall_2_df,
  cols = -Subject,
  names_to = 'condition',
  values_to = 'score')
#> # A tibble: 30 x 3
#>   Subject condition score
#>   <chr>   <chr>     <dbl>
#> 1 Faye   Cued_Neg    15
#> 2 Faye   Cued_Neu    16
#> 3 Faye   Cued_Pos    14
#> 4 Faye   Free_Neg    13
#> 5 Faye   Free_Neu    13
#> 6 Faye   Free_Pos    12
#> 7 Jason Cued_Neg     4
#> 8 Jason Cued_Neu     9
#> 9 Jason Cued_Pos    10
#> 10 Jason Free_Neg     6

```

```
#> # ... with 20 more rows
```

Here, the `condition` is not exactly a variable, but a combination of variables. To `pivot_longer` into two variables, we use two names in `names_to`, and used `names_pattern` to indicate how to split the names `Cued_Neg`, `Cued_Neu`, etc.

```
recall_2_long <- pivot_longer(recall_2_df,
                              cols = -Subject,
                              names_to = c('cue', 'emotion'),
                              names_pattern = '(Cued|Free)_(Neg|Pos|Neu)',
                              values_to = 'score')
```

```
recall_2_long
#> # A tibble: 30 x 4
#>   Subject cue    emotion score
#>   <chr>   <chr> <chr>   <dbl>
#> 1 Faye    Cued  Neg     15
#> 2 Faye    Cued  Neu     16
#> 3 Faye    Cued  Pos     14
#> 4 Faye    Free  Neg     13
#> 5 Faye    Free  Neu     13
#> 6 Faye    Free  Pos     12
#> 7 Jason   Cued  Neg     4
#> 8 Jason   Cued  Neu     9
#> 9 Jason   Cued  Pos    10
#> 10 Jason  Free  Neg     6
#> # ... with 20 more rows
```

To perform the inverse of the above `pivot_longer`, we primarily just need to indicate two columns to take the names from.

```
pivot_wider(recall_2_long,
            names_from = c('cue', 'emotion'),
            values_from = 'score')
#> # A tibble: 5 x 7
#>   Subject Cued_Neg Cued_Neu Cued_Pos Free_Neg Free_Neu Free_Pos
#>   <chr>     <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1 Faye         15     16     14     13     13     12
#> 2 Jason         4      9     10      6      7      9
#> 3 Jim           7      9     10      8      9      5
#> 4 Ron          17     18     20     12     14     15
#> 5 Victor       16     13     14     12     13     14
```

## References

“CrowdFlower 2016 Data Science Report.” 2016. <https://visit.figure-eight.com/data-science-report.html>.

“CrowdFlower 2017 Data Scientist Report.” 2017. <https://www.figure-eight.com/download-2017-data-scientist-report>.

“For Big-Data Scientists, ‘Janitor Work’ Is Key Hurdle to Insights.” 2014. <https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html>.