Introduction to R for Libraries

Part 2: Data Exploration

Written by Clarke Iakovakis. Webinar hosted by ALCTS, Association for Library Collections & Technical Services

May, 2018

Contents

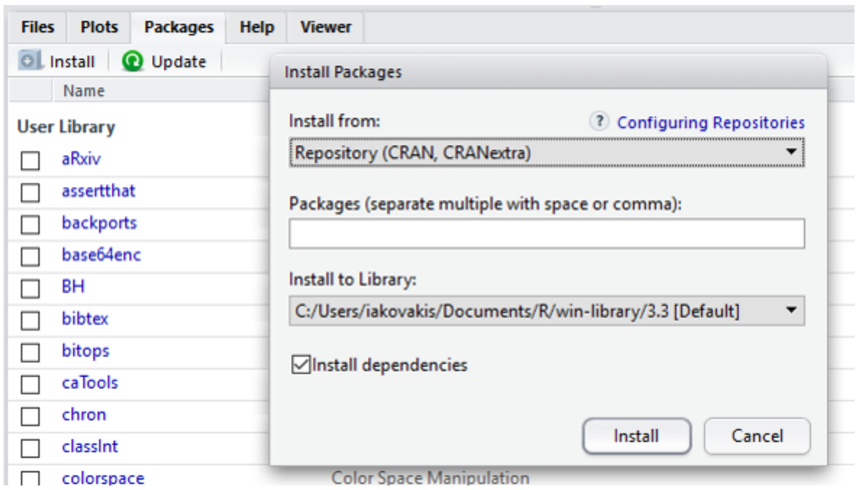
1 Packages

As we have seen, when you download R it already has a number of functions built in: these encompass what is called **Base R.** However, many R users write their own **libraries** of functions, package them together in **R Packages**, and provide them to the R community at no charge. This extends the capacity of R and allows us to do much more. In many cases, they improve on the Base R functions by making them easier and more straightforward to use.

1.1 Installing Packages

The Comprehensive R Archive Network (CRAN) is the main repository for R packages, and that organization maintains strict standards in order for a package to be listed–for example, it must include clear descriptions of the functions, and it must not track or tamper with the user's R session. See this page from RStudio for a good list of useful R packages. In addition to CRAN, R users can make their code and packages available from GitHub. Finally, some communities host their own collections of R packages, such as Bioconductor for computational biology and bioinformatics.

Installing CRAN packages is easy from the RStudio console. Just click the Packages tab in the Navigation Pane, then click Install and search for the package you're looking for. You can also use the `install.packages()` function directly in the console. Run `help(install.packages)` to learn more about how to do it this way.



Install from: Repository (CRAN, CRANextra)

Packages (separate multiple with space or comma):

Install to Library: C:/Users/iakovakis/Documents/R/win-library/3.3 [Default]

[✓] Install dependencies

1.2 Tidyverse

We are going to use three packages in the coming sessions: `dplyr`, `stringr`, and `ggplot2`. You can install each one, one-by-one, or you can install the Tidyverse, which includes all of them and more. `Tidyverse` is actually a package of packages, including many that you are likely to use as you come to work with R.

```r
# install a package. Sometimes you will have to include the argument
# dependencies = TRUE
install.packages("tidyverse")

# If tidyverse is giving you trouble, just install these packages one
# by one:
install.packages("dplyr")
install.packages("readxl")
install.packages("stringr")
install.packages("ggplot2")
```

NOTE: Many campuses have anti-virus running that can interfere with installing packages. You might see the error message "Warning: unable to move temporary installation." If that is the case, try installing the package directly from R (as opposed to RStudio–you can find it in your Applications). For me, this solution on Stack Overflow works best. See also https://stackoverflow.com/a/23167680.

1.3 Loading and using a package

After you install a package, you have to load it into your R session. You can do this using the `library()` function.

```r
# If you try to run a function from a package without loading the
# library, you'll get an error message 'could not find function'
filter(x, z = TRUE)
## Error: could not find function 'filter'

# First, load the package using library
library("dplyr")
```

To see what packages you have installed, and to read more about the functions in the package, click on the Packages tab in the Navigation Pane in RStudio, then click on the package. You can also use the `help` function to get help on a package. Some packages have what are called **vignettes**, which show examples of a package in use.

```r
# get help with a package
help(package = "dplyr")

# see vignettes available for a package. this pops a window up showing
# 5 vignettes for dplyr
vignette(package = "dplyr")

# view a specific vignette
vignette("dplyr")
```

2 Getting data into R

2.1 Ways to get data into R

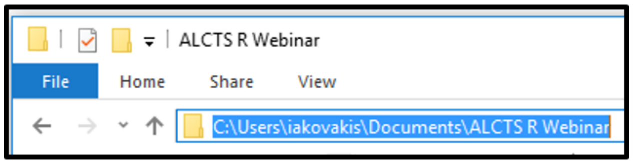
In order to use your data in R, you must import it and turn it into an R *object*. There are many ways to get data into R.

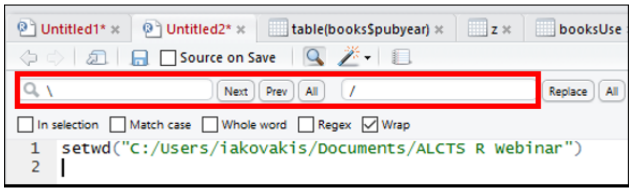
- **Manually**: You can manually create it as we did at the end of last session. To create a data.frame, use the `data.frame()` and specify your variables.
- **Import it from a file** Below is a very incomplete list
  - Text: TXT (`readLines()` function)
  - Tabular data: CSV, TSV (`read.table()` function or `readr` package)
  - Excel: XLSX (`xlsx` package)
  - Google sheets: (`googlesheets` package)
  - Statistics program: SPSS, SAS (`haven` package)
  - Databases: MySQL (`RMySQL` package)
- **Gather it from the web**: You can connect to webpages, servers, or APIs directly from within R, or you can create a data scraped from HTML webpages using the `rvest` package.
  - For example, connect to the Twitter API with the `twitteR` package, or Altmetrics data with `rAltmetric`, or World Bank's World Development Indicators with `WDI`.

2.2 Set your working directory

The **working directory** is the location on your computer R will use for reading and writing files. Use `getwd()` to print your current working directory to the console. Use `setwd()` to set your working directory. There are two important points to make here;

- On Windows computers, directories in file paths are separated with a backslash `\`. However, in R, you must use a forward slash `/`. I usually copy and paste from the Windows Explorer (or Mac Finder) window directly into R and use the find/replace (Ctrl/Cmd + F).
- The directory must be in quotation marks.





```
setwd("C:/Users/iakovakis/Documents/ALCTS R Webinar")
```

```r
# set working directory using a forward slash /
setwd("C:/Users/iakovakis/Documents/ALCTS R Webinar")

# print working directory to the console
getwd()
## [1] 'C:/Users/iakovakis/Documents/ALCTS R Webinar'

# the list.files() function will print all files and folders to the
# console
list.files()
## [1] 'code' 'data' 'doc'
```

Now, you can use period-slash `./` to represent the working directory. So `"./data"` is the same as `"C:/Users/iakovakis/Documents/ALCTS R Webinar/data"`

2.3 Using read.table or read.csv

The `read.table()` function to read a local data file into your R session.

One thing I learned early on is, **always** read your data setting `stringsAsFactors` to `FALSE`. Because R treats factors in a way that can be cumbersome, it is better to coerce it to a factor later if you need to.

Make sure and call `help(read.table)` to read about the important arguments this function takes. This includes:

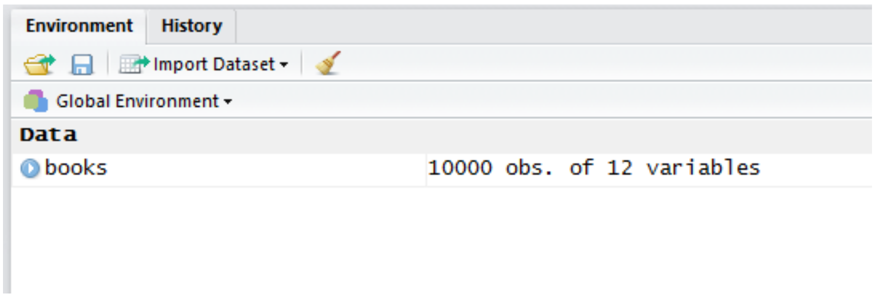
- **`sep = ""`:** Specify the character separating the fields. If you use `read.csv()`, sep is set to a comma by default. I often export data from my ILS with a ^ because there are so many commas in book titles that including more commas can corrupt the data integrity. In that case, I set `sep = "^"`
- **`header = FALSE`:** Set this to true if your data includes headers
- **`colClasses = NA`:** Use this to coerce a variable to a particular **data type** (or class). This is especially important when reading in ISBNs, as R will drop the leading zero if it is read in as numeric. Example: `colClasses = c("ISBN" = "character")`
- **`na.strings = NA`:** Use this to tell R how NA values are represented in your original dataset. For example, if you have blank cells in your original data, you want to set it to empty quotation marks: `na.strings = ""`. Or if your missing data is "N/A" you would set

You can read a CSV (comma-separated values) file into R using the `read.csv()` function. This is the same as `read.table`, except it sets the `sep` to a comma by default. Call `help(read.csv)` to read about the arguments this function takes.

TRY IT YOURSELF

```
# set your working directory to wherever you saved the data files for this webinar
setwd("C:/Wherever you saved the files/ALCTS R Webinar")

# read the books file into your R environment and assign it to books
# not all of these arguments are necessary, but I'm putting them here for example purp
books <- read.csv("./data/raw/books.csv"
                  , sep = ","
                  , na.strings = "NA"
                  , header = TRUE
                  , colClasses
                  = c("SUBJECT" = "character"
                                 , "TOT.CHKOUT" = "integer")
                  , stringsAsFactors = F)
```

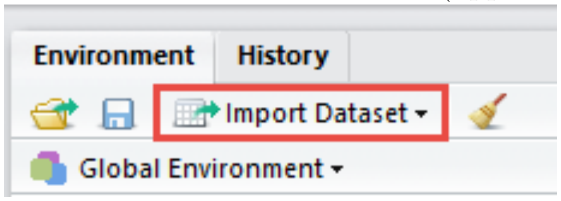
You should now have an R object called `books` in the Environment pane: 4000 observations of 12 variables. We will be using this data file in the next module.



Data
books 10000 obs. of 12 variables

2.4 Import Dataset tool in R Studio

R Studio has a data importing wizard in the Environment Pane (upper right): **Import Dataset:**



I never use this. I type the `read.table()` function directly in my script files, as it gives me more control over the arguments and I am not limited to the choices R Studio gives me.

3 Data exploration

After you read in the data, you want to examine it not only to make sure it was read in correctly, but also to gather some basic information about it. Here I am working with the data file that was provided to you during the webinar session. Read this file by saving it to your computer, setting your working directory, and typing the expression found in the **TRY IT YOURSELF** exercise at the end of Section 2.

3.1 Exploring dataframes

```
# Look at the data in the viewer
View(books)

# Use dim() to obtain the dimensions
dim(books)
## [1] 4000 12

# print the column names
names(books)

# nrow() is number of rows. Also use dim(books)[1]
nrow(books)
## [1] 4000
ncol(books)
## [1] 12

# Use head() and tail() to get the first and last 6 observations View
# more by adding the n argument
head(books)
head(books, n = 10)
```

`sapply()` is a useful way of running a function on all variables in a vector. This webinar series is not going to cover the `apply` series of functions, but you can learn more by watching this series of videos by Roger Peng. Below, it is used to return the class of each variable.

```
sapply(books, class)
## CALL...BIBLIO.        X245.ab         X245.c       LOCATION
##    "character"    "character"    "character"    "character"
##     TOT.CHKOUT       LOUTDATE        SUBJECT            ISN
##      "integer"    "character"    "character"    "character"
##   CALL...ITEM.  X008.Date.One         BCODE2         BCODE1
##    "character"      "integer"    "character"    "character"
```

3.2 Exploring variables

3.2.1 Dollar sign

The dollar sign `$` is used to distinguish a specific variable (column, in Excel-speak) in a data frame:

```
# print the first six book titles
head(books$title)
## [1] "Theorizing feminism :~parallel trends in the humanities and social sciences /"
## [2] "The race for consciousness /"
## [3] "IEEE transactions on aerospace and electronic systems."
## [4] "Nurses in Nazi Germany :~moral choice in history /"
## [5] "Fear of intimacy /"
## [6] "DISCOS :~District and county statistical data /"

# print the mean number of checkouts
mean(books$TOT.CHKOUT)
## [1] 2.7605
```

3.2.2 `unique()`, `table()`, and `duplicated()`

Use `unique()` to see all the distinct values in a variable:

```
unique(books$LOCATION)
##  [1] "clstk" "cltxd" "clfhd" "cljuv" "clusd" "cljre" "clref" "clua " ...
```

Take that one step further with `table()` to get quick frequency counts on a variable:

```
table(books$LOCATION)
## clcdr clchi clcir clcre clfhd clids cljre cljuv clksk ...
##    11    35     1     3   350     8   136   217     1 ...

# you can use it with relational operators (see Table 1 in Section 4)
# Here we find that 7 books have over 50 checkouts
table(books$TOT.CHKOUT > 50)
## FALSE  TRUE
##  3993     7
```

`duplicated()` will give you the a logical vector of duplicated values.

```
# The books dataset doesn't have much duplication
mydupes <- data.frame(identifier = c("111", "222", "111", "333", "444"),
    birthYear = c(1980, 1940, 1980, 2000, 1960))

mydupes
## identifier birthYear 1 111 1980 2 222 1940 3 111 1980 4 333 2000 5
## 444 1960

# The second 111 is duplicated
duplicated(mydupes$identifier)
## [1] FALSE FALSE TRUE FALSE FALSE

# you can put an exclamation mark before it to get non-duplicated
# values
!duplicated(mydupes$identifier)
## [1] TRUE TRUE FALSE TRUE TRUE

# or run a table of duplicated values
table(duplicated(mydupes$identifier))
## FALSE TRUE 4 1

# which() is also a useful function for identifying the specific
# element in the vector that is duplicated
which(duplicated(mydupes$identifier))
## [1] 3
```

3.3 Exploring missing values

You may also need to know the number of missing values:

```
# How many total missing values?
sum(is.na(books))
## [1] 5591

# Total missing values per column
colSums(is.na(books))

# use table() and is.na() in combination
table(is.na(books$isbn))
## FALSE TRUE 2893 1107

# Return only observations that have no missing values
booksNoNA <- na.omit(books)
```

TRY IT YOURSELF

1. Call `View(books)` to examine the data frame.

- Use the small arrow buttons in the variable name to sort TOT.CHKOUT by the highest checkouts. What item has the most checkouts?
- Use the search bar on the right to search for the term Mathematics.
- Click the Filter button above the first variable. Filter the X008.Date.One variable to see only books published after 1960.

2. What is the class of the TOT.CHKOUTS variable?
3. The publication date variable is represented with X008.Date.One. Use the `table()` function to get frequency counts of this variable
4. Use `table()` and `is.na()` to find out how many NA values are in the ISN variable.
5. Use `which()` and `is.na()` to find out which rows are NA in the ISN variable. What happened?
6. Use `which()` and `!is.na()` to find out which rows are **not** NA in ISN. (`!is.na()` is the same thing as `complete.cases()`)
7. Refer back to Section 4.3 in the *Nuts & Bolts of R* handout for Part 1: "Subsetting vectors." Subset the books$ISN vector using `!is.na()` to include only those values that are not NA.
8. Call `summary(books$TOT.CHKOUT)`. What can we infer when we compare the mean, median, and max?
9. `hist()` will print a rudimentary histogram, which displays frequency counts. Call `hist(books$TOT.CHKOUT)`. What is this telling us?

4 Logical tests

R contains a number of operators you can use to compare values. Use `help(Comparison)` to read the R help file. Note that **two equal signs** (`==`) are used for evaluating equality (because one equals sign (`=`) is used for assigning variables).

operator	function.
<	Less Than
>	Greater Than
==	Equal To
<=	Less Than or Equal To
>=	Greater Than or Equal To
!=	Not Equal To
%in%	Has A Match In
is.na	Is NA
!is.na	Is Not NA

Table 1: Relational Operators in R

Sometimes you need to do multiple logical tests (think Boolean logic). Use `help(Logic)` to read the help file.

operator	function.
&	boolean AND
|	boolean OR
!	boolean NOT
any	ANY true
all	ALL true

Table 2: Logical Operators in R

TRY IT YOURSELF

1. Evaluate the following expressions and consider the results.

```r
8 == 8
8 == 16
8 != 16
8 < 16
8 == 16 - 8
8 == 8 & 8 < 16
8 == 8 | 8 > 16
8 == 9 | 8 > 16
any(8 == 8, 8 == 9, 8 == 10)
all(8 == 8, 8 == 9, 8 == 10)
```

```r
8 %in% c(6, 7, 8)
8 %in% c(5, 6, 7)
!(8 %in% c(5, 6, 7))
if (8 == 8) {
    print("eight equals eight")
}
if (8 > 16) {
    print("eight is greater than sixteen")
} else {
    print("eight is less than sixteen")
}
```

5 Data cleaning & transformation with dplyr

We are now entering the data cleaning and transforming phase. While it is possible to do much of the following using Base R functions (in other words, without loading an external package) `dplyr` makes it much easier. Like many of the most useful R packages, `dplyr` was developed by http://hadley.nz/, a data scientist and professor at Rice University.

5.1 Renaming variables

It is often necessary to rename variables to make them more meaningful. If you print the names of the sample `books` dataset you can see that some of the vector names are not particularly helpful:

```r
# print names of the books data frame to the console
names(books)
## [1] 'CALL...BIBLIO.' 'title' 'X245.c' 'LOCATION' 'TOT.CHKOUT'
## 'LOUTDATE' 'SUBJECT' 'ISN' 'CALL...ITEM.'  'X008.Date.One' 'BCODE2'
## 'BCODE1'
```

There are many ways to rename variables in R, but I find the `rename()` function in the `dplyr` package to be the easiest and most straightforward. The new variable name comes first. See `help(rename)`.

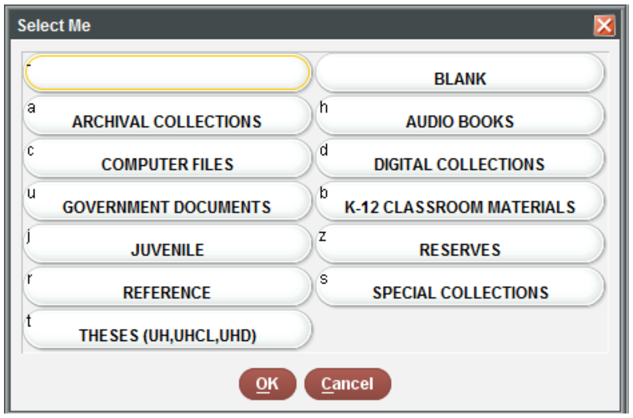
```r
# rename the X245.ab variable. Make sure you return (<-) the output to your
# variable, otherwise it will just print it to the console
books <- rename(books
                , title = X245.ab)

# rename multiple variables at once
books <- rename(books
                , author = X245.c
                , callnumber = CALL...BIBLIO.
                , isbn = ISN
                , pubyear = X008.Date.One
                , subCollection = BCODE1
                , format = BCODE2)
```

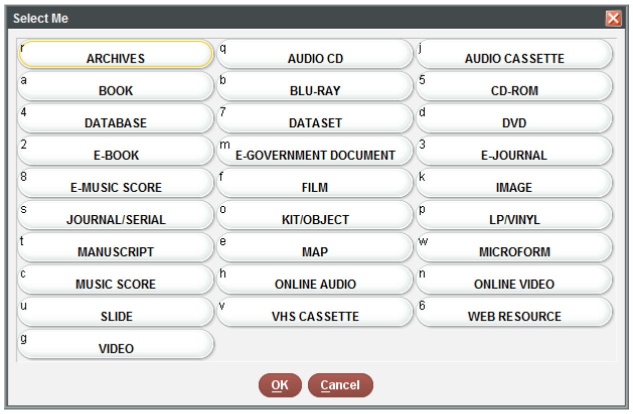
Side note: where does `X245.ab` come from? That is the MARC field 245|ab. However, because R variables cannot start with a number, R automatically inserted and X, and because pipes | are not allowed in variable names, R replaced it with a period. R does this automatically when you read data in, but if you ever need to force it, there's a function `make.names()` that will coerce character vectors to "syntactically valid" names.

```r
make.names(c("245|ab", "CALL #(BIBLIO)"))
## [1] 'X245.ab' 'CALL...BIBLIO.'
```

5.2 Recoding values

It is often necessary to recode or reclassify values in your data. For example, in the sample dataset provided to you, the `subCollection` (formerly `BCODE1`) and `format` (formerly `BCODE2`) variables contain single characters





You can do this easily using the `recode()` function, also in the `dplyr` package.

```r
# first print to the console all of the unique values you will need to recode
unique(books$subCollection)
##   [1] "u" "-" "j" "r" "b" "a" "s" "c" "t" "z"

# Use the recode function to assign them.
# Unlike rename, the old value comes first here.
books$subCollection <- recode(books$subCollection
                              , "-" = "general collection"
                              , u = "government documents"
                              , r = "reference"
                              , b = "k-12 materials"
                              , j = "juvenile"
                              , s = "special collections"
                              , c = "computer files"
                              , t = "theses"
                              , a = "archives"
                              , z = "reserves")
```

TRY IT YOURSELF

1. Run `unique(books$format)` to see the unique values in the `format` column
2. Use `recode()` to rename the `format` values according to the following codes. You **must** put 5 and 4 in quotation marks.

value	name
a	book
e	serial
w	microform
s	e-gov doc
o	map
n	database
k	cd-rom
m	image
"5"	kit/object
"4"	online video

Table 3: Formats

5.3 Subsetting dataframes

5.3.1 Subsetting using brackets in Base R

In the same way we used brackets to subset vectors, we also use them to subset dataframes. However, vectors have only one direction, but dataframes have two. Therefore we have to use two values in the brackets: the first representing the row, and the second representing the column: `[row, column]`

```
# subsetting a vector
c("do", "re", "mi", "fa", "so") [1]

# subsetting a data frame:
# return the second value in the second column
books[5, 2]
## [1] "Fear of intimacy /"

# return the second value in the "titles" column
books[2, "title"]
## [1] "Fear of intimacy /"
```

```
# return the second value in multiple columns
books[2, c("title", "format", "subCollection")]
##                            title format      subCollection
## 2 The race for consciousness /   book general collection

# leave the row space blank to return all rows. This will actually create
# a character vector of all titles.
# This is the equivalent of myTitles <- books$title
myTitles <- books[, "title"]
```

You can also use the relational operators (see above) to return values based on a logical condition. Below, we use `complete.cases()` to subset out the NA values. This function evaluates each row and checks if it is an NA value. If it is not, a TRUE value is returned, and vice versa. Since the column space is blank in the brackets `[row, column]`, it will return all columns.

```
completeCallnumbers <- books[complete.cases(books$callnumber), ]
```

5.3.2 Subsetting using `filter()` in the dplyr package

Subsetting using brackets is important to understand, but as with other R functions, the `dplyr` package makes it much more straightforward, using the `filter()` function.

```
# filter books to return only those items where the format is books
booksOnly <- filter(books, format == "book")

# use multiple filter conditions,
# e.g. books to include only books with more than zero checkouts
bookCheckouts <- filter(books
                        , format == "book"
                        , TOT.CHKOUT > 0)

# How many are there?
# What percentage of all books have at least one checkout?
nrow(bookCheckouts)
## [1] 1968
nrow(bookCheckouts)/nrow(booksOnly) * 100
## [1] 63.93762
# 63%...not bad :-)

library(stringr)
# use the str_detect() function from the stringr package to return all
# books with the word "Science" in the SUBJECT variable, published after 1999
scienceBooks <- filter(books
                       , format == "book"
                       , str_detect(SUBJECT, "Science")
                       , pubyear > 1999)

# We will use stringr more next session.
```

TRY IT YOURSELF

1. Run `unique(format)` and `unique(subCollection)` to confirm the values in each of these fields.
2. Create a data frame consisting only of `format` books. Use the `table()` function to see the breakdown by `subCollection`. Which sub-collection has the most items with 20 or more checkouts?
3. Create a data frame consisting of `format` books and `subCollection` juvenile materials. What is the average number of checkouts `TOT.CHKOUT` for juvenile books?
4. Filter the data frame to include books with between 10 and 20 checkouts

---

5.4 Selecting variables

The `select()` function allows you to keep or remove specific variables. It also provides a convenient way to reorder variables.

```
# specify the variables you want to keep by name
booksTitleCheckouts <- select(books, title, TOT.CHKOUT)

# specify the variables you want to remove with a -
books <- select(books, -CALL...ITEM.)

# reorder columns, combined with everything()
booksReordered <- select(books, title, TOT.CHKOUT, LOUTDATE, everything())
```

5.5 Ordering data

The `arrange()` function in the `dplyr` package allows you to sort your data by alphabetical or numerical order.

```
booksTitleArrange <- arrange(books, title)

# use desc() to sort a variable in descending order
booksHighestChkout <- arrange(books, desc(TOT.CHKOUT))
```

```r
# order data based on multiple variables (e.g. sort first by checkout,
# then by publication year)
booksChkoutYear <- arrange(books, desc(TOT.CHKOUT), desc(pubyear))
```

5.6 Creating new variables

The `mutate()` function allows you to create new variables.

```r
# use the str_sub() function from the stringr package to extract the
# first character of the callnumber variable (the LC Class)
library(stringr)
booksLC <- mutate(books, lc.class = str_sub(callnumber, 1, 1))
```

5.7 Putting it all together with %>%

The pipe operator `%>%` allows you to combine multiple `dplyr` operations simultaneously.

```r
myBooks <- books %>% filter(format == "book") %>% select(title, TOT.CHKOUT) %>%
    arrange(desc(TOT.CHKOUT))
```

TRY IT YOURSELF

Experiment with different combinations of the pipe operator.

1. Create a data frame with these conditions:

- filter to include subCollection juvenile & k-12 materials and format books
- select only title, call number, total checkouts, and pub year
- arrange by total checkouts in descending order

2. Create a data frame with these conditions:

- rename the isbn column to all caps: ISBN
- filter out NA values in the call number column
- filter to include only books published after 1990
- arrange from oldest to newest publication year

5.8 Help with dplyr

- Read more about `dplyr` at https://dplyr.tidyverse.org/.
- In your console, after loading `library(dplyr)`, run `vignette("dplyr")` to read an extremely helpful explanation of how to use it.
- See the http://r4ds.had.co.nz/transform.html in Garrett Grolemund and Hadley Wickham's book *R for Data Science*.
- Watch this Data School video: https://www.youtube.com/watch?v=jWjqLW-u3hc