LEARNING THE TIDYVERSE

AN ILLUSTRATED GUIDE

tidyverse

www.rstudio.com

BY COLLEEN O'BRIANT

CHAPTER 1: INTRODUCTON

CHAPTER 2: TIDIED DATA

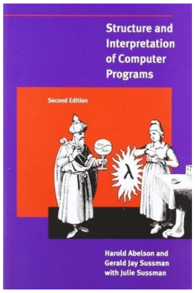
CHAPTER 3: DPLYR

CHAPTER 4: GGPLOT2

CHAPTER 5: PURRR

1 INTRODUCTON

THERE'S ONLY ONE CORRECT WAY TO BEGIN A BOOK ON PROGRAMMING, SO I'LL STEAL THE INTRO FROM HERE:


Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

We are about to study the idea of a *computational process*. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called *data*. The evolution of a process is directed by a pattern of rules called a *program*. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.

A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can answer questions. It can affect the world by disbursing money at a bank or by controlling a robot arm in a factory. The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks we want our processes to perform.

A computational process, in a correctly working computer, executes programs precisely and accurately. Thus, like the sorcerer's apprentice, novice programmers must learn to understand and to anticipate the consequences of their conjuring. Even small errors (usually called *bugs* or *glitches*) in programs can have complex and unanticipated consequences.

Fortunately, learning to program is considerably less dangerous than learning sorcery, because the spirits we deal with are conveniently contained in a secure way. Real-world programming, however, requires care, expertise, and wisdom. A small bug in a computer-aided design program, for example, can lead to the catastrophic collapse of an airplane or a dam or the self-destruction of an industrial robot.

Master software engineers have the ability to organize programs so that they can be reasonably sure that the resulting processes will perform the tasks intended. They can visualize the behavior of their systems in advance. They know how to structure programs so that unanticipated problems do not lead to catastrophic consequences, and when problems do arise, they can *debug* their programs. Well-designed computational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that the parts can be constructed, replaced, and debugged separately.

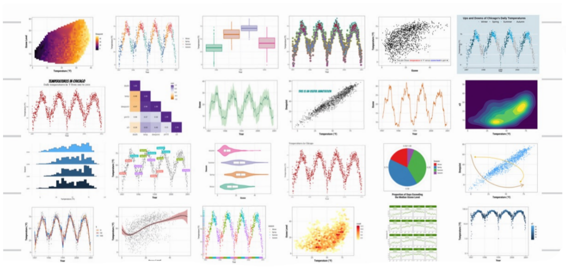
R'S BENEFITS

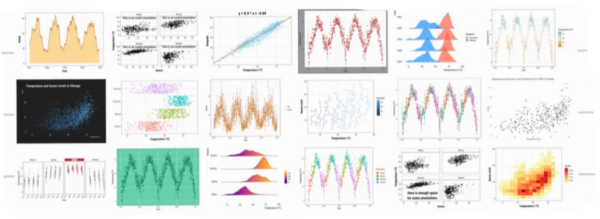
THERE ARE TONS OF LANGUAGES
WE COULD BE LEARNING. WHY R?

1. R'S SYNTAX IS SIMPLE AND IF YOU APPROACH IT CORRECTLY, YOU CAN EASILY GO FROM NO PROGRAMMING SKILLS TO DOING SOPHISTICATED STUFF REALLY FAST.

This is one great advantage of Lisp-like languages: They have very few ways of forming compound expressions, and almost no syntactic structure. All of the formal properties can be covered in an hour, like the rules of chess. After a short time we forget about syntactic details of the language (because there are none) and get on with the real issues—figuring out what we want to compute, how we will decompose problems into manageable parts, and how we will work on the parts.

2. GGPLOT2 MADE THE TIDYVERSE AN INDUSTRY STANDARD FOR DATA ANALYTICS: NOTHING ELSE COMES CLOSE





3. ALL PROGRAMMING LANGUAGES ARE BASICALLY THE SAME, SO JUST LIKE THE ADVICE "THE BEST DIET IS THE ONE YOU STICK WITH", THE BEST PROGRAMMING LANGUAGE IS THE ONE THAT'S SIMPLE AND EASY ENOUGH TO MASTER. ONCE YOU LEARN ONE LANGUAGE WELL, LEARNING OTHERS IS 1000 TIMES EASIER.

4. AND FINALLY, USING R IS JUST GREAT FUN. I HOPE YOU'LL AGREE: IT'S EASY TO USE R TO CREATE ELEGANT + BEAUTIFUL SOLUTIONS TO ALL KINDS OF COMPLICATED DATA ANALYTICS PROBLEMS. AND THAT'S POWERFUL AND WONDERFUL.

R'S SHORTCOMINGS

R IS A VERY POWERFUL GENERAL PURPOSE DATA ANALYTICS ENGINE. BUT YOU MIGHT HEAR PEOPLE SAY THINGS LIKE "R IS SLOW". SO WHAT'S GOING ON? IS IT A GOOD LANGUAGE OR NOT?

HERE'S WHAT I THINK: BEFORE YOU START WRITING A PROGRAM, YOU HAVE A VAGUE IDEA OF HOW YOU'D LIKE TO APPROACH SOLVING THE PROBLEM. AS YOU'RE WRITING IT, YOU'RE FORCED TO FORMALIZE THOSE IDEAS AND MAKE EACH STEP EXPLICIT. BY THE TIME YOU'VE FINISHED THE PROGRAM, YOU UNDERSTAND THE PROBLEM AT ANOTHER LEVEL. THIS IS THE INTELLECTUAL TASK OF PROGRAMMING, AND R IS AT LEAST AS GOOD AS ANY OTHER LANGUAGE AT HELPING YOU DO THIS PART.

BUT SOMETIMES YOU MIGHT HAVE SCALABILITY PROBLEMS: R STARTS SLOWING DOWN OR CRASHING (THIS WON'T BE AN ISSUE IN THIS COURSE OR EVEN MAJOR, BUT YOU MIGHT SEE IT AT YOUR FUTURE JOB). FOR INSTANCE, R DOES GREAT WITH DATASETS UP TO ABOUT A COUPLE MILLION ROWS. IF YOU NEED TO SCALE TO DATASETS WITH A COUPLE BILLION ROWS, YOU CAN RE-IMPLEMENT YOUR SOLUTION IN RUST/JAVA/C. IF YOU NEED TO SCALE TO DATASETS WITH A COUPLE TRILLION ROWS, YOU CAN USE SPARK OR FLINK TO DISTRIBUTE THE PROCESS AMONG SEVERAL COMPUTERS.

R IS A GREAT CHOICE FOR WRITING THAT FIRST PROGRAM THAT DOES THE HARD INTELLECTUAL TASK OF SOLVING THE PROBLEM AT HAND. AFTER YOU'VE DONE THAT, IF YOU FIND R IS TOO SLOW FOR WHAT YOU NEED TO DO, YOU CAN JUST RE-IMPLEMENT YOUR SOLUTION WITH THE BEST TECHNOLOGY FOR THAT USE CASE.

WHAT IS GOOD CODE?

GOOD CODE IS ...

1. CODE THAT GETS YOU THE RIGHT ANSWER!

IF YOU GET THE RIGHT ANSWER, YOU CAN ALWAYS JUST LEAVE IT THERE.

IF YOU HAVE ERRORS, IT'S NOT READY TO TURN IN. KEEP WORKING AND COME TO OFFICE HOURS.

```
Error: unexpected ')'
```

↓

2. CODE THAT SOLVES THE PROBLEM IN THE SIMPLEST POSSIBLE WAY.

LEARN ABOUT FUNCTIONS BY READING THEIR HELP DOCS.

```
> ?qelp::library
> ?qelp::install.packages
```

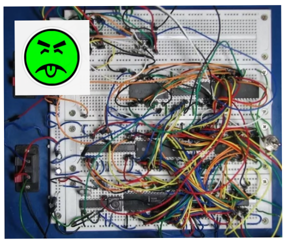
NOT BY ASKING THESE RESOURCES. THEY MAY HELP YOU GET THE ANSWER, BUT THEY WON'T HELP YOU DEVELOP YOUR UNDERSTANDING.

↓

3. CODE THAT IS CLEAR AND READABLE FOR OTHERS

"Programs must be written for people to read, and only incidentally for machines to execute."

— Harold Abelson, Structure and Interpretation of Computer Programs



↓

4. CODE THAT INCLUDES COMMENTS THAT EXPLAIN <u>WHY</u> YOU'RE DOING WHAT YOU'RE DOING.

ANSWER "WHY", NOT "WHAT".

90% of all code comments:

2

TIDIED DATA

2.1 vectors

2.2 pipes

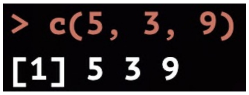
2.3 tibbles

2.1 Vectors

IN R, DATA IS HELD IN VECTORS. YOU CAN CONSTRUCT A VECTOR USING THE FUNCTION `c()`. C IS SHORT FOR "COMBINE": YOU CAN COMBINE ELEMENTS TO FORM A VECTOR.

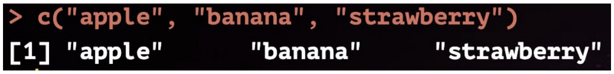
EXAMPLES:

1. COMBINE NUMBERS TO FORM A NUMERIC VECTOR:



```
> c(5, 3, 9)
[1] 5 3 9
```

2. COMBINE CHARACTER STRINGS TO FORM A CHARACTER VECTOR:



```
> c("apple", "banana", "strawberry")
[1] "apple"        "banana"       "strawberry"
```

NOTE: CHARACTER STRINGS NEED QUOTES AROUND THEM. NUMBERS SHOULD NOT HAVE QUOTES AROUND THEM.

YOUR TURN! COMPLETE K01_vectors.R NOW. YOU'LL LEARN:

- MATH OPERATORS WORK ON VECTORS: `+ - / * ^`
- `min()` AND `sum()` WORK ON VECTORS
- USE `length()` TO FIND THE NUMBER OF ELEMENTS IN A VECTOR
- REPEAT ELEMENTS USING `rep()`
- DO RANDOM SAMPLING FROM A VECTOR USING `sample()`

2.2 pipes

THE PIPE: `%>%` IS THE MOST FREQUENTLY USED FUNCTION IN THE TIDYVERSE.

HERE'S WHAT IT DOES:

SUPPOSE YOU HAVE SOME DATA `x` AND YOU'D LIKE TO APPLY SOME FUNCTION `f` ON IT. SO YOU RUN `f(x)`.

FOR EXAMPLE, TAKE THE VECTOR 1:3 AND FIND ITS MINIMUM BY APPLYING min():

```
> min(1:3)
[1] 1
```

ANOTHER WAY TO DO THE SAME THING IS TO USE A PIPE:

```
x %>% f()
```

```
> 1:3 %>% min()
[1] 1
```

THE PIPE SIMPLY TAKES THE DATA THAT COMES BEFORE IT AND INSERTS IT $\longrightarrow$ INTO THE FUNCTION THAT COMES AFTER.

THE WAY YOU SHOULD READ THE PIPE IS WITH THE WORD "THEN", AS IN: "TAKE X, <u>THEN</u> APPLY F."

THERE'S NO REASON TO STOP THERE: WHAT IF WE WANTED TO TAKE x, THEN APPLY F, THEN APPLY G, THEN APPLY H? USING PIPES:

```
x %>% f() %>% g() %>% h()
```

OR USING MULTIPLE LINES, THE PIPE MUST GO AT THE END OF THE LINE:

```
x %>%
  f() %>%
  g() %>%
  h()
```

VERSUS WITHOUT USING PIPES, YOU'D HAVE TO READ INSIDE-OUT.

```
h(g(f(x)))
```

ALL 3 OF THESE WILL OUTPUT THE SAME THING.

YOUR TURN! COMPLETE K02_pipe.R NOW. YOU'LL GET PRACTICE PIPING DATA THROUGH MULTIPLE FUNCTIONS AND COMBINING THE PIPE WITH A PERIOD . TO PIPE DATA INTO THE SECOND ARGUMENT OF A FUNCTION INSTEAD OF THE FIRST.

TO BE CONTINUED ...