UC SANTA BARBARA
Bren School of Environmental
Science & Management

EDS 221: SCIENTIFIC PROGRAMMING ESSENTIALS

by Allison Horst

Contents

Chapter 1

Scientific programming essentials for environmental data science

Material disclaimer and use

This book was created by Allison Horst for EDS 221 (Scientific Programming Essentials) in the Bren School's 1-year Master of Environmental Data Science program at UC Santa Barbara. It accompanies lecture, computational lab and discussion materials that may or may not be linked to throughout the book. This book is intended as a supplemental resource for some parts of the course. In other words, it is not intended as a standalone textbook.

Thank you in advance for suggestions and corrections, which can be submitted as issue to this GitHub repo.

Acknowledgements

I create my courses while standing on shoulders of generous teaching and developing giants in R, data science, and education communities. The wealth and quality of open educational resources (OERs) in data science has made teaching in the field fun, innovative, and inspiring. I've tried to thoroughly credit authors resources that I have pulled from and adapted for this book, and I welcome additions if I have missed any that should be included.

1.1 Course introduction

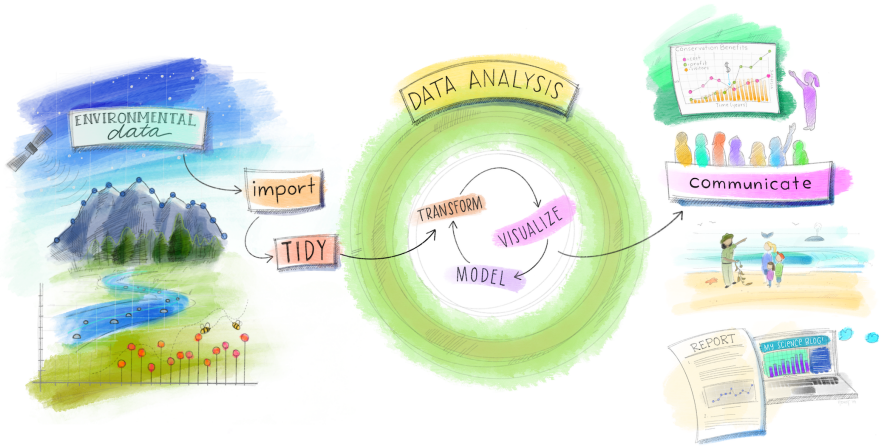

Figure 1.1: Slide from Dr. Julia Lowndes' 2019 keynote talk at useR conference (illustration by Allison Horst).

As nicely summarized in the title of a 2018 NCEAS post, **"the next generation of environmental scientists are data scientists"**. Over the next year in MEDS you'll build skills to responsibly apply advanced methods in environmental modeling, spatial data analysis, machine learning, and more to investigate, analyze and communicate with complex environmental data.

To get there, you'll need a strong foundation in programming basics like: understanding types and structures of data, basic data wrangling and visualization, algorithm development with functions, loops, and conditionals, and how to troubleshoot. While working in the weeds of programming, we'll also learn and reinforce transferable habits for reproducible workflows, robust file paths, version control, data organization, project management, and more.

In EDS 221 you'll also start building versatility by learning fundamental programming skills in different languages (R, Python) and integrated development environments (IDEs) like RStudio and PyCharm, while documenting our work in R Markdown and Jupyter Notebooks.

Upon the building blocks established in EDS 221, you'll be prepared to incrementally grow your advanced environmental data science toolkit throughout MEDS, then enter the workplace at the leading edge of quantitative methods in the field.

1.2 Links to other course materials

- Syllabus
- Code of Conduct
- Course website

1.3 Course setup

We will use the following in EDS 221. You should have pre-installed recent versions before starting the course.

- R (Version 4.0.2 "Taking Off Again", or higher)
- RStudio Desktop (version 1.4.1103 "Wax Begonia", or higher)
- RStudio Cloud (create a free account here)
- Python (version XXXXX or higher)
- Pycharm (version XXXX or higher)
- Jupyter ()

1.4 Course resources

Chapter 2

Meet the 221 tools

2.1 R

2.2 RStudio

2.3 Python

2.4 Jupyter Notebooks

2.5 Playing nicely together

Chapter 3

Data structures and types

How we work with data is largely dependent on the *type* and *structure* of data we're working with. That's more than just "is this a number or letters?" We need to understand how the data are stored so that we know how to access pieces of it, how our code will understand and interact with data, and so we're correctly predicting how data will change based on what we do with it.

In this chapter, we'll learn about different structures of data, and the types of data they can contain.

3.1 Data structures

3.1.1 Vectors

3.1.2 Tibbles

3.1.3 Matrices

3.1.4 Lists

3.2 Data types

See: https://r4ds.had.co.nz/vectors.html

Chapter 4

Basic wrangling

Indexing, subsetting, etc.

Chapter 5

Iteration

From Miriam-Webster Dictionary:

"**Iteration:** (*noun*) the action or a process of iterating or repeating, such as:

- a procedure in which repetition of a sequence of operations yields results successively closer to a desired result
- the repetition of a sequence of computer instructions a specified number of times or until a condition is met"

5.1 Iteration in programming

In programming, iteration is repeating instructions. Usually it's to spare yourself from having to manually do a repetitious thing. Well-written iteration can also make code more readable, usable, and efficient (definitely to write, sometimes to run).

For example, let's consider a few scenarios that may prompt you to use iteration:

- Your data contains 382 columns (variables), and you want to find the mean and standard deviation for each variable
- You have 250 csv files and you want to read them all in and combine them into a single data frame
- In a single data frame you have annual observations for fish passage from 1970 - 2019 recorded at 25 dams in Oregon, and you want to create and save a single graph for passage at each dam

...so basically, anything where you're like "Welp, I guess I'm going to be doing the same thing over and over and over and over..." should inspire you to consider iteration.

5.1.1 A real-world example of iteration in environmental data science

5.2 Generic for loop structure

When we iterate in code, most often that means we're writing some version of a for loop, which we can read as "For these elements in this thing, do this thing to each and return the output, then move on to the next element until you reach the end or a stopping point." There are a bunch of variations on that, but that's the overarching idea.

For example, in the image below our vector is a parade of friendly monsters getting passed through a for loop. There are conditions within the for loop dictating which type of accessory each monster will get, based on their shape. Then the outcome is returned with `print()`.

In words, how can we describe what's happening in this for loop? As each monster is passed individually through the loop, **if it is a triangle**, then it gets sunglasses added to it – that's why the first element in the output vector is a triangle monster with sunglasses. Then we move on to the second (orange) monster. Since they are also a triangle, they're assigned sunglasses. However, when we get to the **third** (purple) monster, it is **not** a triangle, and anything monster shape other than a triangle is assigned a **hat** - so we see the third output is the purple circle monster with a hat.

...and so on until all elements have been passed through the for loop or a stopping point is otherwise reached.

5.3 A basic for loop in R and Python

Let's take a look at some basic for loops, written in both R and Python.

5.3.0.1 Example 1: A vector of very good dogs

Here's our scenario: starting with a vector of dog names "Teddy", "Khora", and "Waffle", write a for loop that returns the statement "[dog name here] is a very good dog!"

In R:

```r
# Create the vector of names:
dog_names <- c("Teddy", "Khora", "Waffle")

# Run it through the loop:
for (i in dog_names) {
```

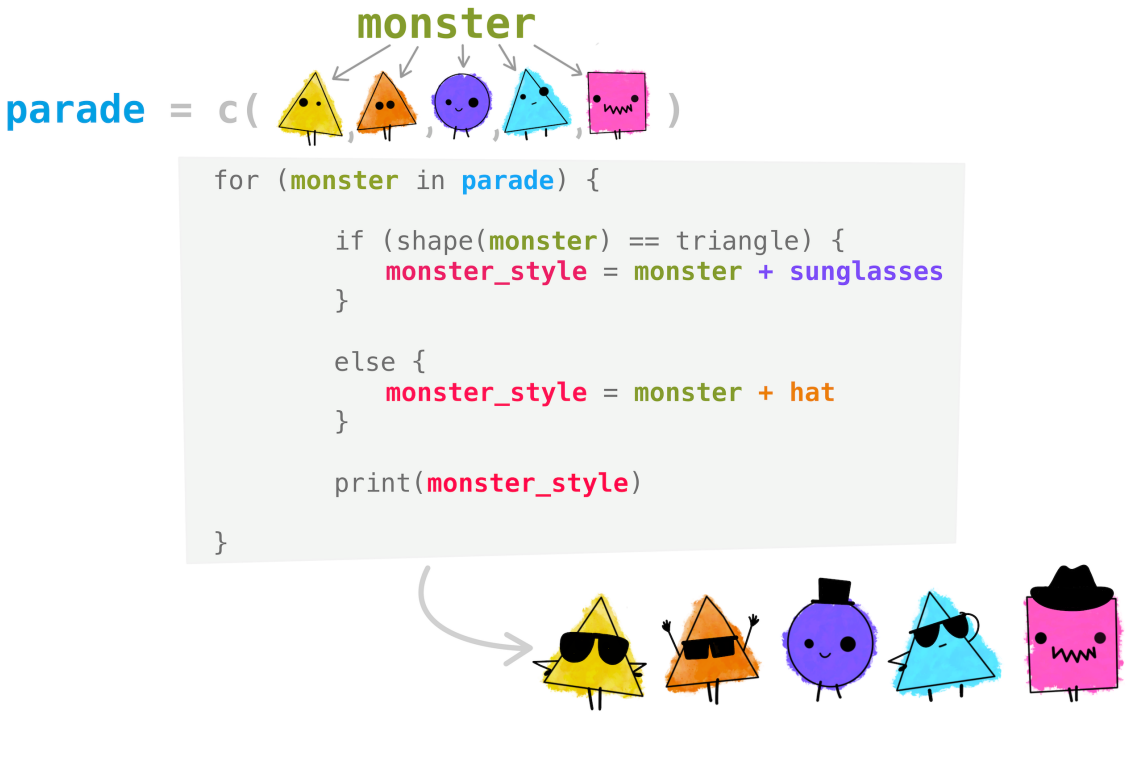

Figure 5.1: Monsters passing through a for loop, getting assigned sunglasses or a hat based on their shape.

```
  print(paste(i, "is a very good dog!"))
}
```

```
## [1] "Teddy is a very good dog!"
## [1] "Khora is a very good dog!"
## [1] "Waffle is a very good dog!"
```

In Python:

```
# Create the vector of names:
dog_names = ['Teddy', 'Khora', 'Waffle']

# Run it through the for loop:
for i in dog_names:
    print(i + " is a very good dog!")
```

```
## Teddy is a very good dog!
## Khora is a very good dog!
## Waffle is a very good dog!
```

5.3.0.2 Example 2: Hypotenuses!

For a vector of values `2, 3, 4, 5, 6, 7`, for any two sequential values, find the length of the hypotenuse if the two values are the lengths of sides of a right triangle. In other words, we'll find the hypotenuse length for right triangles with side lengths 2 & 3, 3 & 4, 4 & 5, etc.

Recall the Pythagorean theorem:

$$a^2 + b^2 = c^2$$

In R:

```
# Make the vector of values:
triangle_sides <- c(2, 3, 4, 5, 6, 7)

# Create the loop to calculate the hypotenuses:
for (i in 1:(length(triangle_sides) - 1)) {
  hypotenuse = sqrt(triangle_sides[i]^2 + triangle_sides[i + 1]^2)
  print(hypotenuse)
}
```

```
## [1] 3.605551
## [1] 5
## [1] 6.403124
## [1] 7.81025
## [1] 9.219544
```

In Python:

Recall: Python indexing starts at ZERO (i.e., the first element in a vector is referenced with `vec[0]`), and the syntax for raising something to a power is `**` (e.g. `x**2`).

A weird one: the `range()` function in Python "...returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops **before** a specified number." So to create a sequence 0, 1, 2, 3, in Python you can use `range(4)`.

```
# Import math library (sqrt() function is not native in Python):
import math

# Make the vector of values:
triangle_sides = [2, 3, 4, 5, 6, 7]

# Create the loop to calculate the hypotenuses:
index_no = range(0, len(triangle_sides) - 1)

for i in index_no:
  hypotenuse = math.sqrt(triangle_sides[i]**2 + triangle_sides[i + 1]**2)
  print(hypotenuse)
```

```
## 3.605551275463989
## 5.0
## 6.4031242374328485
## 7.810249675906654
## 9.219544457292887
```

```
# Aside re: range...this returns sequence 0, 1, 2, 3
demo_vec = range(4)
for i in demo_vec:
  print(i)
```

```
## 0
## 1
## 2
## 3
```

Chapter 6

Conditionals

Chapter 7

Logicals

Chapter 8

Functions

Writing functions to implement algorithms is a fundamental skill for every environmental data scientist. Functions can reduce repetition, increase efficiency and elegance, and facilitate reuse and sharing. Functions built by other developers will be ingrained into your code, but it's also critical that you can build, test, document, and share **your own** functions.

This chapter covers:

- Function structure
- Writing basic functions
- Nested functions
- Functions with iteration and conditions
- Useful function features
- Testing
- Documentation
- Applied examples

8.1 Function components

At the most basic level, a function takes an input, does something to it (a calculation, transformation, etc.), and returns an output.

For example, we can write a function that doubles the input value. In *function notation* seen in math, that would be:

$$f(x) = 2x$$

where x is the input, and $f(x)$ is the output. The function f acts on input x by doubling the input value.

How can we create a function to do the same thing in R? An R function would look like this:

```
double_it <- function(x) {
  2*x
}
```

What are these different pieces of that function?

- **function name**: Here, the function is named `double_it`
- **formals**: The `function(x)` piece defines the function *formals* (arguments / parameters). This function expects a single input argument, `x` (you can check what the formals are using `formals(function_name)`).
- **body**: here, `{ 2*x }` is the body of the function - that's where we tell it what to do with the inputs. Note the braces (i.e. squiggly brackets) are often on separate lines from the algorithm itself.

Try out the function by inputting both a single value, and a vector of values. Note that vectorization is the default - meaning that the function is applied to each element in a vector.

```
double_it(x = 20)
```

```
## [1] 40
```

```
vec <- c(2, 4, 50) # Create a vector with multiple values

double_it(vec) # Function acts on each element in the vector
```

```
## [1]   4   8 100
```

Those are the main pieces. But don't worry, it gets a lot more interesting. Let's start by writing a few of our own functions.

8.1.1 A note on names

It's important to be thoughtful when naming functions. We generally want to follow standard practices for good names (concise, descriptive, code and coder-friendly), but you may also consider the following: - Start with a verb that describes what the function *does* (e.g. `sort`, `build`, `predict`) - End with a noun describing the thing it works with or creates (e.g. `image`, `model`, `mass`) - Combine them with a coder-friendly case (like `lower_snake_case`)

For example, here are some suggestions that may be useful function names:

`` `sum_imports` ``, `` `predict_offsets` ``, `` `plot_simulations` ``

In contrast, here are some function names that may be less useful, memorable, and intuitive for you and collaborators:

`` `fun_1` ``, `` `calc` ``, `` `x2` ``

Table 8.1: Parameter estimates for selected Hawaiian fish from Peyton et al. (2015).

Scientific name	Common name	a	b
Chanos chanos	Milkfish	0.0905	2.52
Sphyraena barracuda	Great barracuda	0.0181	3.27
Caranx ignobilis	Giant trevally	0.0353	3.05

It is likely that there will be a tradeoff between conciseness and descriptiveness. While there aren't *rules* about naming functions, I recommend erring on the side of descriptiveness to make reading and writing code a bit more intuitive. With tab-completion, the decrease in efficiency is minimal.

8.2 Writing simple functions

Let's practice writing a few simple functions using established relationships in environmental science.

8.2.1 Example 1: Fish standard weight

"Standard weight" is how much we *expect* a fish to weigh, give the species and fish length, and the nonlinear relationship is given by:

$$W = aL^b$$

where L is total fish length (centimeters), W is the expected fish weight (grams), and a and b are species-dependent parameter values.

Write a function to calculate fish weight based on a, b, and fish length, then estimate the weight of several fish based on the following parameter estimates for Hawaiian fish from Peyton et al. (2016):

Function:

```
predict_weight <- function(a, length, b) {
  a*(length^b)
}
```

Using the function:

1. Estimate the mass of a 160 cm long great barracuda.
2. Estimate the mass of a 118 cm long milkfish.

Thinking ahead: Does this pass your smell test for a user-friendly and user-helpful function? How might we make this function simpler for a user? For example, maybe a user can input the *species*, and the parameters a and b can be correctly sourced from a table? We'll learn how to add this kind of functionality in upcoming sections.

8.2.2 Example 2: Wind turbines

The full power in wind hitting a turbine is:

$$P = 0.5\rho A v^3$$

where P is power in Watts (joules/second), ρ is the air density (kg/m^3), A is the area covered by the turbine blades (square meters), and v is the wind velocity (m/s).

However, the Betz Limit means that turbines can only collect ~60% of the total wind power, which updates the theoretical "collectable" power (before accounting for inefficiencies, losses, etc.) to:

$$P = 0.3\rho A v^3$$

Write a function to calculate *maximum collectable* wind power (Watts) by a turbine requiring three inputs:

- Air density (in kg/m^3)
- Rotor radius (in meters)
- Wind velocity (in m/s)

Write the function:

```
calc_windpower <- function(rho, radius, windspeed) {

  0.3*rho*pi*(radius^2)*(windspeed^3)

}
```

Can we clean this up a bit by calculating the area first, within the function? Sure!

```
calc_windpower <- function(rho, radius, windspeed) {

  # Calculate turbine area (meters squared):
  turbine_area = pi*(radius^2)

  # Calculate collectable power:
```

```
  0.3*rho*turbine_area*(windspeed^3)
}
```

Now let's use the function we've created.

The largest turbine in the world (as of March 2021) is the GE Haliade-X, an offshore turbine prototype in Rotterdam, the Netherlands, with a 220 meter rotor diameter.

Assuming a windspeed of 7.7 m/s (based on long-term averages for North sea North Sea platforms from Coelingh et al. (1998)) and an air density of 1.225 kg/m^3 (at sea level), estimate the wind power that can be collected.

```
calc_windpower(rho = 1.225, radius = 110, windspeed = 7.7) # Watts
```

```
## [1] 6377710
```

8.3 Functions with conditionals

In the examples above, we change input values, but what the function *does* doesn't change based on those input values.

Sometimes, we'll want our function to do something different (e.g. a different calculation, use a different constant value) based on the input values.

8.4 Functions with iteration

8.5 Useful function features

8.6 Testing functions

8.7 Iterating functions

8.8 Resources on building, testing, & documenting functions

- Ch. 6 - Functions in *Advanced R* by Hadley Wickham

Chapter 9

Tidy data

Chapter 10

Data wrangling & viz in the tidyverse

Chapter 11

Troubleshooting

Bibliography

Coelingh, J., van Wijk, A., and Holtslag, A. (1998). Analysis of wind speed observations on the North Sea coast. *Journal of Wind Engineering and Industrial Aerodynamics*, 73(2):125–144.

Peyton, K. A., Sakihara, T. S., Nishiura, L. K., Shindo, T. T., Shimoda, T. E., Hau, S., Akiona, A., and Lorance, K. (2016). Length–weight relationships for common juvenile fishes and prey species in Hawaiian estuaries. *Journal of Applied Ichthyology*, 32(3):499–502.