Strings and Regular Expressions

Grinnell College

February 19, 2026

Warm-up

Pretty sure I have that join warm-up to do

Today

Goals for today:

1. General R Housekeeping
2. Strings and processing
3. Regular expressions

The theme for the day is: "Strings are data, and regular expressions are a tool for describing patterns in data"

Housekeeping

The two primary types of objects we have worked with in R are **vectors** and **data.frames**.

Vectors represent the most basic unit in R, and are defined by a collection of items that are all of the same "type". This "type" determines the class of the vector.

The hierarchical ordering of vectors is:

- Logical
- Numeric (integer then double)
- Character

This hierarchy comes from the idea of *coercion*; vectors of a lower hierarchical type can be represented by a higher type, but not vice versa

Housekeeping

We can tell the class of an object by using the `class()` function

```
> x <- c("a", "b", "c")
> y <- c(TRUE, FALSE)
> class(x)
[1] "character"
> class(y)
[1] "logical"
```

At their core, a data.frame is essentially a collection of vectors, forced to be the same length

```
> str(iris)
'data.frame': 150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1
```

Vectorization

Because vectors make up the atomic unit in R, many functions in R are constructed to handle vectors as input

Whereas a language like C++ would require you to loop through each element of a vector to perform an operation, we don't think twice about using functions like this in R:

```
> z <- c(1.4, 2.6, 3.8)
> sqrt(z)
[1] 1.1832 1.6125 1.9494
```

This instance of a function working on a vector is known as **vectorization**

Strings

Strings are data, strings are data, strings are data

- Names or places
- Treatment types
- Metadata, ID numbers, gene types, etc.,

Strings exist as a distinct and totally generalized type of data; anything that can be represented as data can be represented as a string

What we need are tools to manipulate them

Strings as Data

Understanding strings as data requires understanding that the way a computer interacts with strings is different than the way we might understand them. They involve a few particularities:

- 10 != “10”
- ”2” > “10”
- “Male” vs “male” vs “ male ”
- Capitalization, whitespace, special (unicode) characters

With enough experience, the things to look out for become almost a second nature.

String Functions

Just like with dplyr, we can frame most of what we will want to do around a few verbs:

- Detect – Does this pattern exist?
- Extract – Give me the piece I care about
- Replace – Standardize and clean
- Split/combine/mutate – Organize and modify

A non-trivial portion of string manipulation is managing errors from manual entry (e.g., "free response")

Regular Expression

While strings have some basic arithmetic properties (“”2” ¿ “10”, for example), what we need are more versatile ways to express what we want

At its essentials, regular expressions (or regex) are a language for describing patterns in text.

For example

Suppose I want to select all elements that contain *only* the word "cat" or "caat" or "caaat" (with any number of "a"s in between). How would I differentiate from the following?

1. cat
2. caaaat
3. cats
4. cats?
5. concatenate

Searching for the **literal** "cat" will falsely flag each of these entries when we only want the first.

These are the kinds of scenarios where we need to describe patterns

Side Notes

Our goals with regular expressions (today and forever) are to build intuition, not to memorize

Sure we can use literals, but we can also not use literals

Instead, let's think of a few basic pieces and consider how they may be used as building blocks for something more elaborate

6 sins

1. Wildcard .
2. Character classes [A-Z], [0-9], [:punct:]
3. Quantifiers *, +, {n}
4. Anchors, ^, $, \b
5. Grouping (), {}
6. Escape and meta characters

Example

An example using quantifiers and anchoring

```
> x <- c("cat", "caaaat", "cats", "cats?", "concatenate")
>
> str_view(x, pattern = "cat")
[1]  <cat>
[3]  <cat>s
[4]  <cat>s?
[5]  con<cat>enate
>
> str_view(x, pattern = "^ca+t$")
[1]  <cat>
[2]  <caaaat>
```

Motivation

Q: Where might we need to specify a general pattern for string manipulation?
A: Literally everywhere

Common tasks include...

- Extract metadata from filenames
- Find email usernames without the domain
- Extract numbers from text
- Standardize URLs and API calls

Cheating

Two things can both be true:

- Regular expressions can be fun
- They can be really tedious and hard

It's absolutely worthwhile knowing how they work, along with a few basic rules of syntax, but more complicated tasks are a perfect use-case for LLMs