Lecture 11: Working with Strings and Date/Time Variables

Managing and Manipulating Data Using R

Introduction

Logistics

Reading

- GW chapter 14 (Strings)
- GW chapter 16 (Dates and Times)

No class next week (11/13)

- Problem set for strings + date/times still due on 11/13
- No problem set due on 11/20

What we will do today

1. Introduction

2. Working with Strings

   2.1 String basics

   2.2 Regular Expressions

Load the packages we will use today (output omitted)

- **you must run this code chunk after installing these packages**

```
library(tidyverse)
library(stringr)
```

If package not yet installed, then must install before you load. Install in “console” rather than .Rmd file

- Generic syntax: `install.packages("package_name")`
- Install “tidyverse”: `install.packages("stringr")`

Note: when we load package, name of package is not in quotes; but when we install package, name of package is in quotes:

- `install.packages("tidyverse")`
- `library(tidyverse)`

Load data we will use today

- Western Washington University student list data

```
load(url("https://github.com/ozanj/rclass/raw/master/data/prospect_list/wwlist_
```

Working with Strings

String basics

What are strings?

String refers to a “data type” used in programming to represent text rather than numbers (although it can include numbers)

- Strings have `character` types

```
string1<- "Apple"
typeof(string1) #type is charater
#> [1] "character"
```

- Create strings using `" "`

```
string2 <- "This is a string"
```

- If string contains a quotation, use `' " " '`

```
string3 <- 'example of a "quote" within a string'
```

- To print a string, use `writeLines()`

```
print(string3) #will print using \
#> [1] "example of a \"quote\" within a string"
writeLines(string3)
#> example of a "quote" within a string
```

Common uses of strings

Basic uses:

- Names of files and directories

```
acs_tract <- read_csv("https://raw.githubusercontent.com/ozanj/rclass/master/da
#> Warning: Missing column names filled in: 'X1' [1]
#> Parsed with column specification:
#> cols(
#>   .default = col_double(),
#>   tract_name = col_character(),
#>   tract = col_character(),
#>   race_brks_nonwhiteasian = col_character(),
#>   inc_brks = col_character()
#> )
#> See spec(...) for full column specifications.
```

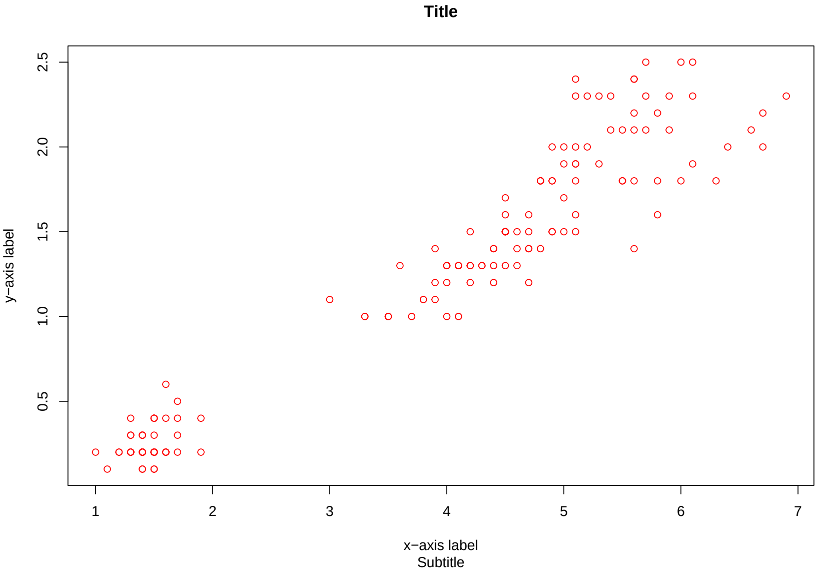
- Names of elements in data objects

```
num_vec <- 1:5
names(num_vec) <- c('uno', 'dos', 'tres', 'cuatro', 'cinco')
num_vec
#>    uno    dos   tres cuatro  cinco
#>      1      2      3      4      5
```

Common uses of strings

- Text elements displayed in plots, graphs, maps

```
plot(iris$Petal.Length, iris$Petal.Width, main = 'Title', sub = 'Subtitle',
     xlab = 'x-axis label', ylab = 'y-axis label', col = 'red')
```



String basics

We will use the `stringr` library for working with strings, rather than Base R

- `stringr` functions have intuitive names and all begin with `str_`
- Base R functions for working with strings can be inconsistent (avoid using them)

Basic functions:

- String length using `str_length()`

```r
#example 1
string2 <- "This is a string"
str_length(string2)
#> [1] 16

#example 2
str_length(c("a", "strings are fun", NA))
#> [1]  1 15 NA
```

Combining strings

- Combining strings using `str_c()`

```r
#example 1
x_var <- "x"
y_var <- "y"

str_c(x_var, y_var)
#> [1] "xy"

#example 2
str_c("x", "y")
#> [1] "xy"
```

- Use `sep` argument to control how strings are seperated when combined

```r
str_c("x", "y", sep= ", ")
#> [1] "x, y"
```

- `NA` are still contagious, if you want a string "NA" rather than `NA` use `str_replace_na()`

```r
street_dir<- c("East", "West", NA)
str_c("Direction: ", street_dir)
#> [1] "Direction: East" "Direction: West" NA
str_c("Direction: ", str_replace_na(street_dir))
#> [1] "Direction: East" "Direction: West" "Direction: NA"
```

Subsetting strings

-Extract parts of a string using `str_sub()` , which uses `start` and `end` arguments to extract the position of the substring wanted

```
fruits<- c("Apple", "Banana", "Orange")

#first three elements
str_sub(fruits, 1, 3) #end argument in inclusive
#> [1] "App" "Ban" "Ora"

#last three elements
str_sub(fruits, -3, -1) #neg nums count backwards from end
#> [1] "ple" "ana" "nge"
```

- **Task: extract 6-digit zip code from** `zip9` **in** `wwlist`

```
wwlist %>% mutate(
  zip=str_sub(zip9, 1, 5)
)
```

Lower-case and Upper-case functions

- Changing strings to lower or upper case

```
str_to_lower("HELLO")
#> [1] "hello"
str_to_upper("hello")
#> [1] "HELLO"
```

- **Task: lower-case `hs_name` in `wwlist`**

```
wwlist %>% select(receive_date, hs_name) %>%
  mutate(
  hs_name_lwr=str_to_lower(hs_name),
)
#> # A tibble: 268,396 x 3
#>    receive_date hs_name                       hs_name_lwr
#>    <date>       <chr>                         <chr>
#>  1 2016-05-31   Ingraham High School          ingraham high school
#>  2 2016-05-31   Kentwood Senior High School   kentwood senior high school
#>  3 2016-05-31   Archbishop Thomas J Murphy HS archbishop thomas j murphy hs
#>  4 2016-05-31   Garfield High School          garfield high school
#>  5 2016-05-31   Lake Stevens High School      lake stevens high school
#>  6 2016-05-31   Franklin High School          franklin high school
#>  7 2016-05-31   Hockinson High School         hockinson high school
#>  8 2016-05-31   Nathan Hale High School       nathan hale high school
#>  9 2016-05-31   Sultan High School            sultan high school
#> 10 2016-05-31   Sandpoint High School         sandpoint high school
#> # ... with 268,386 more rows
```

Other `stringr` Functions

I only highlighted a few `stringr` functions in this lecture. But there are many!

- `stringr` cheat sheet
- Some common functions:

Task	**Function**
Detect matches	`str_detect` , `str_which` , `str_count` , `str_locate`
Subset strings	`str_sub` , `str_subset` , `str_extract` , `str_match`
Mutate strings	`str_sub` , `str_replace` , `str_to_lower` , `str_to_upper`
Join or split strings	`str_c` , `str_dup` , `str_plit_fixed`

Student Exercises

1. Combine `school_type` and `school_category` in the `wwlist` dataframe to create one school type + category varibale. Be sure to seperate type and category using a comma AND deal with contagious NAs by using string “NA” if `school_type` and/or `school_category` are `NA`.
2. The last four digits of `zip9` indicate the delivery route within the 5-digit zip code area. Create a new `route` variable that extracts the last four digits from `zip9`.

Student Excercises (Solutions)

1. Combine `school_type` and `school_category` in the `wwlist` dataframe to create one school type + category varibale. Be sure to seperate type and category using a comma AND deal with contagious NAs by using string “NA” if `school_type` and/or `school_category` are `NA`.

```
wwlist %>% select(school_type, school_category) %>%
  mutate(
    type_cat= str_c(str_replace_na(school_type), str_replace_na(school_category)
  )
#> # A tibble: 268,396 x 3
#>    school_type school_category type_cat
#>    <chr>       <chr>           <chr>
#>  1 public      Regular School  public, Regular School
#>  2 public      Regular School  public, Regular School
#>  3 <NA>        <NA>            NA, NA
#>  4 public      Regular School  public, Regular School
#>  5 public      Regular School  public, Regular School
#>  6 public      Regular School  public, Regular School
#>  7 public      Regular School  public, Regular School
#>  8 public      Regular School  public, Regular School
#>  9 public      Regular School  public, Regular School
#> 10 public      Regular School  public, Regular School
#> # ... with 268,386 more rows
```

Student Excercises (Solutions)

1. The last four digits of `zip9` indicate the delivery route within the 5-digit zip code area. Create a new `route` variable that extracts the last four digits from `zip9`.

```
wwlist %>% select(zip9) %>%
  mutate(
  route=str_sub(zip9, -4, -1)
)
#> # A tibble: 268,396 x 2
#>    zip9       route
#>    <chr>      <chr>
#>  1 98103-3528 3528
#>  2 98030-7964 7964
#>  3 98290-8659 8659
#>  4 98105-0002 0002
#>  5 98252-9327 9327
#>  6 98108-1809 1809
#>  7 98685-3135 3135
#>  8 98125-4543 4543
#>  9 98294-1529 1529
#> 10 83864-2304 2304
#> # ... with 268,386 more rows
```

Regular Expressions

What are regular expressions (e.g., regex)?

Regular expressions are an entirely different and consise “language” used to describe patterns in strings

- One of the most powerful and sophisticated data science tools!
- They have a wide range of uses
- They are universal: can be used and are consistent across any programming language (e.g., R, Python, C/C+,)
- **BUT** they take a while to wrap your head around and can get really complex really quickly!

I will attempt to give an approachable introduction to regular expressions

- I still stuggle with regular expression tasks!
- My favorite tool for building, testing, debugging regular expressions: web regex app

Why are string manipulations, e.g., regular expressions, useful?

Some real word examples:

- Dealing with identification numbers (leading or trailing zeros)

```
typeof(acs_tract$fips_county_code)
#> [1] "double"

acs_tract <- acs_tract %>%
  mutate(char_county=
  str_pad(as.character(fips_county_code), side = "left" ,3, pad="0"))
```

- Matching valid street addresses

Common uses for regular expressions

-Complex reshaping (tidying) of data

- Problem: multiple variables crammed into the column names
  - new_ prefix = new cases
  - sp/rel/sp/ep describe how the case was diagnosed
  - m/f gives the gender
  - digits are age ranges

```r
who %>% pivot_longer(
  cols = new_sp_m014:newrel_f65,
  names_to = c("diagnosis", "gender", "age"),
  names_pattern = "new_?(.*)_(.)(.*)",
  values_to = "count"
)
```