

# Lecture 11: Working with Strings and Date/Time Variables

## Managing and Manipulating Data Using R

# Introduction

## 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
3. Working with Dates and Times
  - 3.1 Creating Date/Times
  - 3.2 Using Date/Time Variables

## Load the packages we will use today (output omitted)

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

```
library(tidyverse)
library(stringr)
library(lubridate)
library(nycflights13)
```

**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_
```

- ▶ Jeopardy text data [Link](#)

```
jeopardy <- read_csv("https://raw.githubusercontent.com/ozanj/rclass/master/data/jeopardy.csv")  
#> Parsed with column specification:  
#> cols(  
#>   `Show Number` = col_double(),  
#>   `Air Date` = col_date(format = ""),  
#>   Round = col_character(),  
#>   Category = col_character(),  
#>   Value = col_character(),  
#>   Question = col_character(),  
#>   Answer = col_character()  
#> )
```

## 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 character  
#> [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/data/acs/tract.csv")
#> 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.
```

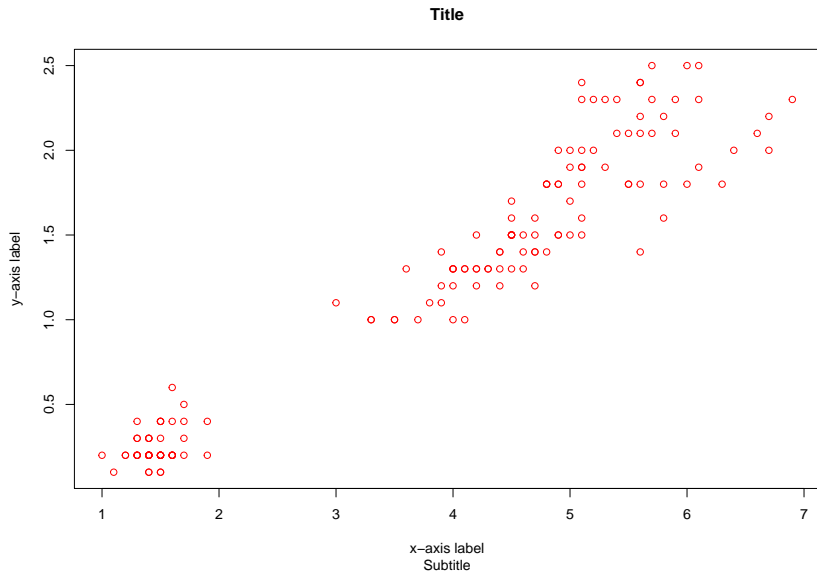
## ► Names of elements in data objects

```
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()`

*#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()`

```
#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 separated when combined

```
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()`

```
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 is 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
```

## Student Exercises

1. Combine `school_type` and `school_category` in the `wwlist` dataframe to create one school type + category variable. Be sure to separate 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 Exercises (Solutions)

1. Combine `school_type` and `school_category` in the `wwlist` dataframe to create one school type + category variable. Be sure to separate 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 Exercises (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 concise “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, JavaScript)
- ▶ **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 struggle with regular expression tasks!
- ▶ My favorite tool for building, testing, debugging regular expressions: [web regex app](#)

## Basic Matches

The simplest patterns match exact (sub)strings!

- ▶ `str_view()` shows the first match; `str_view_all` shows all the matches

```
x <- c("apple", "banana", "pear")  
# str_view(x, "an") #uncomment to view outside of beamer presentation
```

- ▶ To detect matches in a column of a dataframe, use `str_detect` and `filter()`
  - ▶ `str_detect` determines a match and returns a logical vector the same length as the input

**Task:** Detect whether high school names abbreviate “high school” as “HS”?

```
wwlist %>%  
  select(hs_name) %>%  
  filter(str_detect(hs_name, "HS"))  
#> # A tibble: 9,072 x 1  
#>   hs_name  
#>   <chr>  
#> 1 Archbishop Thomas J Murphy HS  
#> 2 Lewis and Clark HS-Spokane  
#> 3 Health Sci & Human Services HS  
#> 4 Lewis and Clark HS-Spokane  
#> 5 Cascade HS - Leavenworth  
#> 6 Auburn Mountainview HS  
#> 7 Newport HS- Bellevue  
#> 8 Newport HS- Bellevue  
#> 9 East Valley HS-Spokane Valley  
#> 10 Marysville Pilchuck Pathway HS
```

## Basic Matches

The next step-up in complexity is using `.` which matches any character (including white space but except a newline)

**Task:** Detect whether there are any “HS” abbreviations that have *any* character before and after the abbreviation?

```
wwlist %>%  
  select(hs_name) %>%  
  filter(str_detect(hs_name, ".HS."))  
#> # A tibble: 4,919 x 1  
#>   hs_name  
#>   <chr>  
#> 1 Lewis and Clark HS-Spokane  
#> 2 Lewis and Clark HS-Spokane  
#> 3 Cascade HS - Leavenworth  
#> 4 Newport HS- Bellevue  
#> 5 Newport HS- Bellevue  
#> 6 East Valley HS-Spokane Valley  
#> 7 Liberty HS-Renton  
#> 8 East Valley HS-Yakima  
#> 9 West Valley HS-Spokane  
#> 10 Newport HS- Bellevue  
#> # ... with 4,909 more rows
```

# Anchors

Regular expressions will match any part of a string. Sometimes it's useful to *anchor* the regular expression so that it matches from the start or the end of the string:

- ▶ `^` will match the start of the string
- ▶ `$` will match the end of the string

```
x <- c("apple", "banana", "pear")
str_detect(x, "^a")
#> [1] TRUE FALSE FALSE
str_detect(x, "a$")
#> [1] FALSE TRUE FALSE
```

# Escapes

Regular expressions use special characters (i.e., `.`, `\`) to match patterns in strings. If you're trying to match an actual character exactly rather than use its special behavior, you need to use an *escape*

► The double backslash `\\` is used to *escape* special behavior for these characters

```
# To create the regular expression, we need \\  
dot <- "\\."
```

```
# But the expression itself only contains one:  
writeLines(dot)  
#> \.
```

```
# And this tells R to look for an explicit .  
str_detect(c("abc", "a.c", "bef"), "a\\.c")  
#> [1] FALSE TRUE FALSE
```

► for a full list of escape characters type the following into your console: `?\"'`



## Common special patterns

There are other “special patterns” that will match more than one character and can be really useful.

### Matching characters

- ▶ `.` matches any character
- ▶ `\d` matches any digit (or `\D` for non-digits)
- ▶ `\s` matches any whitespace such as space, tab, newline (or `\S` for non-whitespace)

### Matching alternates

- ▶ `[abe]` matches one of a, b, or e
- ▶ `[^ab3]` matches anything but a, b, e
- ▶ `[a-f]` matches range

## Tools: other `stringr` Functions

I only highlighted a few `stringr` functions in this lecture. But there are many functions that are helpful in applying regular expressions to real data problems (i.e., determining match, finding positions of matches, extracting context of matches, replacing values based on matches)

► [stringr cheat sheet](#)

► Some common functions:

Task	Function
Detect matches	<code>str_detect</code> , <code>str_which</code> , <code>str_count</code> , <code>str_locate</code>
Subset strings	<code>str_sub</code> , <code>str_subset</code> , <code>str_extract</code> , <code>str_match</code>
Mutate strings	<code>str_sub</code> , <code>str_replace</code> , <code>str_to_lower</code> , <code>str_to_upper</code>
Join or split strings	<code>str_c</code> , <code>str_dup</code> , <code>str_split_fixed</code>

## Examples

**Task:** Look for any Jeopardy categories that begin with a digit

```
jeopardy %>%  
  select(Category) %>%  
  filter(str_detect(Category, "^\\d"))  
#> # A tibble: 6,294 x 1  
#>   Category  
#>   <chr>  
#> 1 3-LETTER WORDS  
#> 2 3-LETTER WORDS  
#> 3 3-LETTER WORDS  
#> 4 3-LETTER WORDS  
#> 5 3-LETTER WORDS  
#> 6 1994 FILMS  
#> 7 1994 FILMS  
#> 8 1994 FILMS  
#> 9 1994 FILMS  
#> 10 1994 FILMS  
#> # ... with 6,284 more rows
```

## Examples

**Task:** Look for any Jeopardy categories that may contain a year variable as a sequence of 4 digits. Create a new `year_category` indicator variable if the Jeopardy category involves a year.

```
jeopardy1 <- jeopardy %>%  
  select(Category) %>%  
  mutate(  
    year_category=str_detect(Category, "[0-9][0-9][0-9][0-9]")  
  )
```

► Print some observations

```
jeopardy1 %>%  
  select(Category, year_category) %>%  
  filter(year_category==TRUE)  
#> # A tibble: 3,347 x 2  
#>   Category      year_category  
#>   <chr>         <lgl>  
#> 1 1994 FILMS TRUE  
#> 2 1994 FILMS TRUE  
#> 3 1994 FILMS TRUE  
#> 4 1994 FILMS TRUE  
#> 5 1994 FILMS TRUE  
#> 6 THE 1930s TRUE  
#> 7 THE 1930s TRUE  
#> 8 THE 1930s TRUE  
#> 9 THE 1930s TRUE  
#> 10 THE 1930s TRUE
```

# Why are string manipulations and regular expressions useful?

Basic 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"))
```

- ▶ 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

```
who %>% pivot_longer(
```

```
  cols = new_sp_m014:newrel_f65,
```

```
  names_to = c("diagnosis", "gender", "age"),
```

```
  names_pattern = "new_?(.*)_(.)(.*)",
```

```
  values_to = "count"
```

```
)
```

# Why are string manipulations and regular expressions useful?

## Advanced examples:

- ▶ Web-scraping
  - ▶ Find and scrape all linked pages of recruiters assigned by states:  
(<https://gobama.ua.edu/staff/>)
  - ▶ Parsing raw HTML to convert it into tabular data
- ▶ Natural Language Processing
  - ▶ Analyzing university president speeches for promotion of interdisciplinary research (IDR)
  - ▶ Predict sentiment of promotion of IDR

## Working with Dates and Times

## Working with date/time variables

Working with dates and times in data management seems simpler than it really is!

- ▶ Does every year have 365 days?
- ▶ Does every day have 24 hours?
- ▶ Does every minute have 60 seconds?

These details matter for:

- ▶ Calculating changes over time
- ▶ Analyzing longitudinal data
- ▶ Predicting the occurrence/timing of events

There are three ways you're likely to create a date/time variable:

- ▶ From a string (most common)
- ▶ From date and time individual components
- ▶ From an existing date/time object



## Creating Date/Times

## Creating Date/Times from strings

The most common way you're likely to create Date/Time variables is from primary/secondary data where dates and times are recorded and/or stores as strings.

For Dates:

- ▶ Use `lubridate` "helpers" to identify the order of year/month/day

```
ymd("2017/01/31")
```

```
#> [1] "2017-01-31"
```

```
mdy("January 31st, 2017")
```

```
#> [1] "2017-01-31"
```

```
dmy("31-01-2017")
```

```
#> [1] "2017-01-31"
```

```
ymd(20170131)
```

```
#> [1] "2017-01-31"
```

For Dates:

- ▶ Use `lubridate` "helpers" to identify the order of year/month/day **AND** hours/minutes/seconds

```
ymd_hms("2017-01-31 20:11:59")
```

```
#> [1] "2017-01-31 20:11:59 UTC"
```

```
mdy_hm("01/31/2017 08:01")
```

```
#> [1] "2017-01-31 08:01:00 UTC"
```

## Creating Date/Times from individual variables

What if your dates and times are recorded across multiple columns/variables?

EX: NYC flights data

```
flights %>%  
  select(year, month, day, hour, minute)  
#> # A tibble: 336,776 x 5  
#>   year month   day hour minute  
#>   <int> <int> <int> <dbl> <dbl>  
#> 1  2013     1     1     5     15  
#> 2  2013     1     1     5     29  
#> 3  2013     1     1     5     40  
#> 4  2013     1     1     5     45  
#> 5  2013     1     1     6      0  
#> 6  2013     1     1     5     58  
#> 7  2013     1     1     6      0  
#> 8  2013     1     1     6      0  
#> 9  2013     1     1     6      0  
#> 10 2013     1     1     6      0  
#> # ... with 336,766 more rows
```

► Create a date variable using `make_date()`

```
flights1<- flights %>%  
  select(year, month, day) %>%  
  mutate(  
    depart= make_date(year, month, day)
```

## Using Date/Time Variables

# Time spans

Arithmetic with dates works differently than with any numeric type!

There are three date/time classes that represent time spans:

- ▶ Durations: represent the duration of time to an exact number of seconds
- ▶ Periods: represent the period of time such as weeks/months/years
- ▶ Intervals: represent a starting and end point in time

## Durations

When you subtract two dates, the result is a `difftime` object

- ▶ A `difftime` object records time span as seconds (not intuitive)
- ▶ Use `as.duration` to make the `difftime` object more intuitive (but records time span in seconds)

```
# How old is Karina?  
k_age <- today() - ymd(19890321)  
k_age  
#> Time difference of 11187 days  
  
typeof(k_age)  
#> [1] "double"  
class(k_age)  
#> [1] "difftime"  
  
as.duration(k_age)  
#> [1] "966556800s (~30.63 years)"
```

- ▶ Durations uses “constructors”

# Time spans

## Periods

Periods don't record time spans in exact seconds and are more intuitive to the way we think about time!

```
one_pm <- ymd_hms("2016-03-12 13:00:00", tz = "America/New_York")
```

```
one_pm # 1pm  
#> [1] "2016-03-12 13:00:00 EST"  
one_pm + days(1) #1pm  
#> [1] "2016-03-13 13:00:00 EDT"
```

```
one_pm + years(1)  
#> [1] "2017-03-12 13:00:00 EDT"
```