

Data Cleaning

Introduction to R for Public Health Researchers

Data Cleaning

In general, data cleaning is a process of investigating your data for inaccuracies, or recoding it in a way that makes it more manageable.

MOST IMPORTANT RULE - LOOK AT YOUR DATA!

Useful checking functions

- `is.na` - is TRUE if the data is FALSE otherwise
- `!` - negation (NOT)
 - if `is.na(x)` is TRUE, then `!is.na(x)` is FALSE
- `all` takes in a logical and will be TRUE if ALL are TRUE
 - `all(!is.na(x))` - are all values of `x` NOT NA
- `any` will be TRUE if ANY are true
 - `any(is.na(x))` - do we have any NA's in `x`?
- `complete.cases` - returns TRUE if EVERY value of a row is NOT NA
 - very stringent condition
 - FALSE missing one value (even if not important)

Dealing with Missing Data

Missing data types

One of the most important aspects of data cleaning is missing values.

Types of “missing” data:

- `NA` - general missing data
- `NaN` - stands for “**N**ot **a** **N**umber”, happens when you do $0/0$.
- `Inf` and `-Inf` - Infinity, happens when you take a positive number (or negative number) by 0.

Finding Missing data

Each missing data type has a function that returns `TRUE` if the data is missing:

- `NA` - `is.na`
- `NaN` - `is.nan`
- `Inf` and `-Inf` - `is.infinite`
- `is.finite` returns `FALSE` for all missing data and `TRUE` for non-missing

Missing Data with Logicals

One important aspect (esp with subsetting) is that logical operations return NA for NA values. Think about it, the data could be > 2 or not we don't know, so R says there is no TRUE or FALSE, so that is missing:

```
x = c(0, NA, 2, 3, 4, -0.5, 0.2)
x > 2
```

```
[1] FALSE      NA FALSE    TRUE    TRUE FALSE  FALSE
```

Missing Data with Logicals

What to do? What if we want if $x > 2$ and x isn't NA?

Don't do $x \neq \text{NA}$, do $x > 2$ and x is NOT NA:

```
x != NA
```

```
[1] NA NA NA NA NA NA NA
```

```
x > 2 & !is.na(x)
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
```


Missing Data with Logicals

What about seeing if a value is equal to multiple values? You can do `(x == 1 | x == 2) & !is.na(x)`, but that is not efficient.

```
(x == 0 | x == 2) # has NA
```

```
[1]  TRUE      NA  TRUE FALSE FALSE FALSE FALSE
```

```
(x == 0 | x == 2) & !is.na(x) # No NA
```

```
[1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
```

what to do?

Missing Data with Logicals: %in%

Filter removes missing values, have to keep them if you want them:

```
df = tibble(x = x)
df %>% filter(x > 2)
```

```
# A tibble: 2 x 1
      x
  <dbl>
1     3
2     4
```

```
filter(df, between(x, -1, 3) | is.na(x))
```

```
# A tibble: 6 x 1
      x
  <dbl>
1     0
2    NA
3     2
4     3
5   -0.5
6    0.2
```

dplyr::filter

Be careful with missing data using subsetting:

```
x %in% c(0, 2, NA) # this
```

```
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

```
x %in% c(0, 2) | is.na(x) # versus this
```

```
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
```

Missing Data with Operations

Similarly with logicals, operations/arithmetic with NA will result in NAs:

```
x + 2
```

```
[1] 2.0 NA 4.0 5.0 6.0 1.5 2.2
```

```
x * 2
```

```
[1] 0.0 NA 4.0 6.0 8.0 -1.0 0.4
```

Lab Part 1

[Website](#)

Tables and Tabulations

Useful checking functions

- `unique` - gives you the unique values of a variable
- `table(x)` - will give a one-way table of `x`
 - `table(x, useNA = "ifany")` - will have row NA
- `table(x, y)` - will give a cross-tab of `x` and `y`

Creating One-way Tables

Here we will use `table` to make tabulations of the data. Look at `?table` to see options for missing data.

```
unique(x)
```

```
[1]  0.0  NA  2.0  3.0  4.0 -0.5  0.2
```

```
table(x)
```

```
x
-0.5    0  0.2    2    3    4
    1    1    1    1    1    1
```

```
table(x, useNA = "ifany") # will not
```

```
x
-0.5    0  0.2    2    3    4 <NA>
    1    1    1    1    1    1    1
```


Creating One-way Tables

useNA = "ifany" will not have NA in table heading if no NA:

```
table(c(0, 1, 2, 3, 2, 3, 3, 2, 2, 3),  
      useNA = "ifany")
```

```
0 1 2 3  
1 1 4 4
```

Creating One-way Tables

You can set `useNA = "always"` to have it always have a column for NA

```
table(c(0, 1, 2, 3, 2, 3, 3, 2, 2, 3),  
      useNA = "always")
```

0	1	2	3	<NA>
1	1	4	4	0

Tables with Factors

If you use a `factor`, all levels will be given even if no exist! - (May be wanted or not):

```
fac = factor(c(0, 1, 2, 3, 2, 3, 3, 2, 2, 3),  
             levels = 1:4)  
tab = table(fac)  
tab
```

```
fac  
1 2 3 4  
1 4 4 0
```

```
tab[ tab > 0 ]
```

```
fac  
1 2 3  
1 4 4
```

Creating Two-way Tables

A two-way table. If you pass in 2 vectors, `table` creates a 2-dimensional table.

```
tab <- table(c(0, 1, 2, 3, 2, 3, 3, 2, 2, 3),  
             c(0, 1, 2, 3, 2, 3, 3, 4, 4, 3),  
             useNA = "always")  
tab
```

	0	1	2	3	4	<NA>
0	1	0	0	0	0	0
1	0	1	0	0	0	0
2	0	0	2	0	2	0
3	0	0	0	4	0	0
<NA>	0	0	0	0	0	0

Finding Row or Column Totals

`margin.table` finds the marginal sums of the table. `margin` is 1 for rows, 2 for columns in general in R. Here is the column sums of the table:

```
margin.table(tab, 2)
```

	0	1	2	3	4	<NA>
1	1	1	2	4	2	0

Proportion Tables

`prop.table` finds the marginal proportions of the table. Think of it dividing the table by it's respective marginal totals. If `margin` not set, divides by overall total.

```
prop.table(tab)
```

	0	1	2	3	4	<NA>
0	0.1	0.0	0.0	0.0	0.0	0.0
1	0.0	0.1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.2	0.0	0.2	0.0
3	0.0	0.0	0.0	0.4	0.0	0.0
<NA>	0.0	0.0	0.0	0.0	0.0	0.0

```
prop.table(tab,1) * 100
```

	0	1	2	3	4	<NA>
0	100	0	0	0	0	0
1	0	100	0	0	0	0
2	0	0	50	0	50	0
3	0	0	0	100	0	0
<NA>						

Lab Part 2

[Website](#)

Download Salary FY2014 Data

From <https://data.baltimorecity.gov/City-Government/Baltimore-City-Employee-Salaries-FY2015/nsfe-bg53> <https://data.baltimorecity.gov/api/views/nsfe-bg53/rows.csv>

Read the CSV into R `Sal`:

```
Sal = jhur::read_salaries() # or  
Sal = read_csv("http://data.baltimorecity.gov/api/views/nsfe-bg53/rows.csv")  
Sal = rename(Sal, Name = name)
```


Checking for logical conditions

- `any()` - checks if there are any TRUES
- `all()` - checks if ALL are true

```
head(Sal, 2)
```

	Name	JobTitle	AgencyID
1	Aaron, Patricia G	Facilities/Office Services II	A03031
2	Aaron, Petra L	ASSISTANT STATE'S ATTORNEY	A29045

	Agency	HireDate	AnnualSalary	GrossPay
1	OED-Employment Dev (031)	10/24/1979	\$55314.00	\$53626.04
2	States Attorneys Office (045)	09/25/2006	\$74000.00	\$73000.08

```
any(is.na(Sal$Name)) # are there any NAs?
```

```
[1] FALSE
```

Recoding Variables

Example of Recoding

For example, let's say gender was coded as Male, M, m, Female, F, f. Using Excel to find all of these would be a matter of filtering and changing all by hand or using if statements.

In `dplyr` you can use the `recode` function:

```
data = data %>%  
  mutate(gender = recode(gender, M = "Male", m = "Male", M = "Male"))
```

or use `ifelse`:

```
data %>%  
  mutate(gender = ifelse(gender %in% c("Male", "M", "m"),  
                          "Male", gender))
```

Example of Cleaning: more complicated

Sometimes though, it's not so simple. That's where functions that find patterns come in very useful.

```
table(gender)
```

gender									
F	FeMAle	FEMALE	Fm	M	Ma	mAle	Male	MaLe	MALE
75	82	74	89	89	79	87	89	88	95
Man	Woman								
73	80								

Example of Cleaning: more complicated

```
table(gender)
```

```
gender
female Female      fm   male   Male
      156      155    89    359    241
```

Strings functions

Splitting/Find/Replace and Regular Expressions

- R can do much more than find exact matches for a whole string
- Like Perl and other languages, it can use regular expressions.
- What are regular expressions?
 - Ways to search for specific strings
 - Can be very complicated or simple
 - Highly Useful - think “Find” on steroids

A bit on Regular Expressions

- <http://www.regular-expressions.info/reference.html>
- They can use to match a large number of strings in one statement
- `.` matches any single character
- `*` means repeat as many (even if 0) more times the last character
- `?` makes the last thing optional
- `^` matches start of vector `^a` - starts with "a"
- `$` matches end of vector `b$` - ends with "b"

The **stringr** package

The `stringr` package:

- Makes string manipulation more intuitive
- Has a standard format for most functions
 - the first argument is a string like first argument is a `data.frame` in `dplyr`
- We will not cover `grep` or `gsub` - base R functions
 - are used on forums for answers
- Almost all functions start with `str_*`

Let's look at modifier for `stringr`

?modifiers

- `fixed` - match everything exactly
- `regex` - default - uses **regular expressions**
- `ignore_case` is an option to not have to use `tolower`

Substring and String Splitting

- `str_sub(x, start, end)` - substrings from position start to position end
- `str_split(string, pattern)` - splits strings up - returns list!

```
library(stringr)
x <- c("I really", "like writing", "R code programs")
y <- str_split(x, " ") # returns a list
y
```

```
[[1]]
[1] "I"      "really"
```

```
[[2]]
[1] "like"    "writing"
```

```
[[3]]
[1] "R"        "code"     "programs"
```

Using a fixed expression

One example case is when you want to split on a period “.”. In regular expressions `.` means **ANY** character, so

```
str_split("I.like.strings", ".")
```

```
[[1]]  
[1] "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
```

```
str_split("I.like.strings", fixed("."))
```

```
[[1]]  
[1] "I"      "like"    "strings"
```

Let's extract from **y**

```
y[[2]]
```

```
[1] "like"      "writing"
```

```
sapply(y, dplyr::first) # on the fly
```

```
[1] "I"        "like" "R"
```

```
sapply(y, nth, 2) # on the fly
```

```
[1] "really"    "writing" "code"
```

```
sapply(y, last) # on the fly
```

```
[1] "really"    "writing" "programs"
```

Separating columns based on a separator

- From `tidyr`, you can split a data set into multiple columns:

```
df = tibble(x = c("I really", "like writing", "R code programs"))
```

```
df %>% separate(x, into = c("first", "second", "third"))
```

```
Warning: Expected 3 pieces. Missing pieces filled with NA in column 3. Additional pieces dropped in column 2].
```

```
# A tibble: 3 x 3
  first second third
<chr> <chr> <chr>
1 I      really <NA>
2 like   writing <NA>
3 R      code   programs
```

```
# A tibble: 3 x 2
  first second
<chr> <chr>
1 I      really
2 like   writing
3 R      code
```

Separating columns based on a separator

- `extra = "merge"` will not drop data. Also, you can specify the separator

```
df = tibble(x = c("I really", "like. _writing R. But not", "R code programs"))
```

```
df %>% separate(x, into = c("first", "second", "third"), extra = "merge")
```

Warning: Expected 3 pieces. Missing pieces filled with `NA` in 1 rows [1].

```
# A tibble: 3 x 3
  first second third
<chr> <chr>   <chr>
1 I      really <NA>
2 like   writing R. But not
3 R      code   programs
```

Separating columns based on a separator

- `extra = "merge"` will not drop data. Also, you can specify the separator

```
df = tibble(x = c("I really", "like. _writing R. But not", "R code programs"))
```

```
df %>% separate(x, into = c("first", "second", "third"), extra = "merge", sep
```

Warning: Expected 3 pieces. Missing pieces filled with `NA` in 1 rows [1].

```
# A tibble: 3 x 3
  first second  third
<chr> <chr>    <chr>
1 I      really  <NA>
2 like.  _writing R. But not
3 R      code    programs
```


'Find' functions: **stringr**

`str_detect`, `str_subset`, `str_replace`, and `str_replace_all` search for matches to argument pattern within each element of a character vector: they differ in the format of and amount of detail in the results.

- `str_detect` - returns TRUE if pattern is found
- `str_subset` - returns only the strings which pattern were detected
 - convenient wrapper around `x[str_detect(x, pattern)]`
- `str_extract` - returns only strings which pattern were detected, but ONLY the pattern
- `str_replace` - replaces pattern with replacement the first time
- `str_replace_all` - replaces pattern with replacement as many times matched

'Find' functions: Finding Logicals

These are the indices where the pattern match occurs:

```
head(str_detect(Sal$Name, "Rawlings"))
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

'Find' functions: Finding Indices

These are the indices where the pattern match occurs:

```
which(str_detect(Sal$Name, "Rawlings"))
```

```
[1] 10256 10257 10258
```

Showing difference in `str_extract`

`str_extract` extracts just the matched string

```
ss = str_extract(Sal$Name, "Rawling")  
head(ss)
```

```
[1] NA NA NA NA NA NA
```

```
ss[ !is.na(ss) ]
```

```
[1] "Rawling" "Rawling" "Rawling"
```

'Find' functions: finding values, stringr and dplyr

```
str_subset(Sal$Name, "Rawlings")
```

```
[1] "Rawlings,Kellye A"          "Rawlings,Paula M"  
[3] "Rawlings-Blake,Stephanie C"
```

```
Sal %>% filter(str_detect(Name, "Rawlings"))
```

	Name	JobTitle	AgencyID
1	Rawlings,Kellye A	EMERGENCY DISPATCHER	A40302
2	Rawlings,Paula M	COMMUNITY AIDE	A04015
3	Rawlings-Blake,Stephanie C	MAYOR	A01001

	Agency	HireDate	AnnualSalary	GrossPay
1	M-R Info Technology (302)	01/06/2003	\$48940.00	\$73356.42
2	R&P-Recreation (015)	12/10/2007	\$19802.00	\$10443.70
3	Mayors Office (001)	12/07/1995	\$167449.00	\$165249.86

Using Regular Expressions

- Look for any name that starts with:
 - Payne at the beginning,
 - Leonard and then an S
 - Spence then capital C

```
head(str_subset( Sal$Name, "^Payne.*"), 3)
```

```
[1] "Payne El,Boaz L"      "Payne El,Jackie"  
[3] "Payne Johnson,Nickole A"
```

```
head(str_subset( Sal$Name, "Leonard.?S"))
```

```
[1] "Payne,Leonard S"      "Szumlanski,Leonard S"
```

```
head(str_subset( Sal$Name, "Spence.*C.*"))
```

```
[1] "Spencer,Charles A"  "Spencer,Clarence W" "Spencer,Michael C"
```

Showing difference in `str_extract` and `str_extract_all`

`str_extract_all` extracts all the matched strings - `\\d` searches for DIGITS/numbers

```
head(str_extract(Sal$AgencyID, "\\d"))
```

```
[1] "0" "2" "6" "9" "4" "9"
```

```
head(str_extract_all(Sal$AgencyID, "\\d"), 2)
```

```
[[1]]
```

```
[1] "0" "3" "0" "3" "1"
```

```
[[2]]
```

```
[1] "2" "9" "0" "4" "5"
```

Showing difference in `str_replace` and `str_replace_all`

`str_replace_all` extracts all the matched strings

```
head(str_replace(Sal$Name, "a", "j"))
```

[1] "Ajron, Patricia G"	"Ajron, Petra L"	"Abjineh, Yohannes T"
[4] "Abbene, Anthony M"	"Abbey, Emmjnuel"	"Abbott-Cole, Michelle"

```
head(str_replace_all(Sal$Name, "a", "j"), 2)
```

```
[1] "Ajron, Pjtricij G" "Ajron, Petrj L"
```


Replace

Let's say we wanted to sort the data set by Annual Salary:

```
class(Sal$AnnualSalary)
```

```
[1] "factor"
```

```
head(Sal$AnnualSalary, 4)
```

```
[1] $55314.00 $74000.00 $64500.00 $46309.00  
1654 Levels: $10000.00 $100000.00 $100013.00 $100200.00 ... $99994.00
```

```
head(as.numeric(Sal$AnnualSalary), 4)
```

```
[1] 908 1302 1094 722
```

R didn't like the \$ so it thought turned them all to NA.

Replacing and substituting

Now we can replace the \$ with nothing (used `fixed("$")` because \$ means ending):

```
Sal = Sal %>% mutate(  
  AnnualSalary = str_replace(AnnualSalary, fixed("$"), ""),  
  AnnualSalary = as.numeric(AnnualSalary)  
) %>%  
  arrange(desc(AnnualSalary))
```

Pasting strings with `paste` and `paste0`

Paste can be very useful for joining vectors together:

```
paste("Visit", 1:5, sep = "_")
```

```
[1] "Visit_1" "Visit_2" "Visit_3" "Visit_4" "Visit_5"
```

```
paste("Visit", 1:5, sep = "_", collapse = " ")
```

```
[1] "Visit_1 Visit_2 Visit_3 Visit_4 Visit_5"
```

```
paste("To", "is going be the ", "we go to the store!", sep = "day ")
```

```
[1] "Today is going be the day we go to the store!"
```

```
# and paste0 can be even simpler see ?paste0  
paste0("Visit", 1:5)
```

```
[1] "Visit1" "Visit2" "Visit3" "Visit4" "Visit5"
```

Uniting columns based on a separator

- From `tidyr`, you can unite:

```
df = tibble(id = rep(1:5, 3), visit = rep(1:3, each = 5))
```

```
df %>% unite(col = "unique_id", id, visit)
```

```
# A tibble: 15 x 1
```

```
  unique_id  
  <chr>
```

```
1 1_1  
2 2_1  
3 3_1  
4 4_1  
5 5_1  
6 1_2  
7 2_2  
8 3_2  
9 4_2  
10 5_2  
11 1_3  
12 2_3  
13 3_3  
14 4_3  
15 5_3
```

```
# A tibble: 15 x 3
```

```
  unique_id  id visit  
  <chr>    <int> <int>
```

```
1 1_1      1     1  
2 2_1      2     1  
3 3_1      3     1  
4 4_1      4     1  
5 5_1      5     1  
6 1_2      1     2  
7 2_2      2     2  
8 3_2      3     2  
9 4_2      4     2  
10 5_2     5     2  
11 1_3      1     3  
12 2_3      2     3  
13 3_3      3     3  
14 4_3      4     3  
15 5_3      5     3
```

Paste Depicting How Collapse Works

```
paste(1:5)
```

```
[1] "1" "2" "3" "4" "5"
```

```
paste(1:5, collapse = " ")
```

```
[1] "1 2 3 4 5"
```

Useful String Functions

Useful String functions

- `toupper()`, `tolower()` - uppercase or lowercase your data:
- `str_trim()` (in the `stringr` package) or `trimws` in base
 - will trim whitespace
- `nchar` - get the number of characters in a string

Sorting characters

- `sort` - reorders the data - characters work, but not correctly
- `rank` - gives the rank of the data - ties are split
- `order` - gives the indices, if subset, would give the data sorted
 - `x[order(x)]` is the same as sorting

```
sort(c("1", "2", "10")) # not sort correctly (order simply ranks the data)
```

```
[1] "1"  "10" "2"
```

```
order(c("1", "2", "10"))
```

```
[1] 1 3 2
```

```
x = rnorm(10)
x[1] = x[2] # create a tie
rank(x)
```

```
[1] 3.5 3.5 1.0 8.0 5.0 7.0 6.0 9.0 2.0 10.0
```

Lab Part 3

[Website](#)

Website

[Website](#)

Comparison of **stringr** to base R -
not covered

Splitting Strings

Substringing

Very similar:

Base R

- `substr(x, start, stop)` - substrings from position start to position stop
- `strsplit(x, split)` - splits strings up - returns list!

stringr

- `str_sub(x, start, end)` - substrings from position start to position end
- `str_split(string, pattern)` - splits strings up - returns list!

Splitting String: base R

In base R, `strsplit` splits a vector on a string into a list

```
x <- c("I really", "like writing", "R code programs")  
y <- strsplit(x, split = " ") # returns a list  
y
```

```
[[1]]  
[1] "I"      "really"
```

```
[[2]]  
[1] "like"   "writing"
```

```
[[3]]  
[1] "R"      "code"   "programs"
```

Showing difference in `str_extract` and `str_extract_all`

`str_extract_all` extracts all the matched strings - `\\d` searches for DIGITS/numbers

```
head(str_extract(Sal$AgencyID, "\\d"))
```

```
[1] "2" "9" "6" "2" "0" "0"
```

```
head(str_extract_all(Sal$AgencyID, "\\d"), 2)
```

```
[[1]]
```

```
[1] "2" "9" "0" "0" "1"
```

```
[[2]]
```

```
[1] "9" "9" "3" "9" "0"
```

'Find' functions: base R

`grep`: `grep`, `grepl`, `regexpr` and `gregexpr` search for matches to argument pattern within each element of a character vector: they differ in the format of and amount of detail in the results.

`grep(pattern, x, fixed=FALSE)`, where:

- `pattern` = character string containing a regular expression to be matched in the given character vector.
- `x` = a character vector where matches are sought, or an object which can be coerced by `as.character` to a character vector.
- If `fixed=TRUE`, it will do exact matching for the phrase anywhere in the vector (regular find)

'Find' functions: stringr compared to base R

Base R does not use these functions. Here is a “translator” of the `stringr` function to base R functions

- `str_detect` - similar to `grep1` (return logical)
- `grep(value = FALSE)` is similar to `which(str_detect())`
- `str_subset` - similar to `grep(value = TRUE)` - return value of matched
- `str_replace` - similar to `sub` - replace one time
- `str_replace_all` - similar to `gsub` - replace many times

Important Comparisons

Base R:

- Argument order is `(pattern, x)`
- Uses option `(fixed = TRUE)`

`stringr`

- Argument order is `(string, pattern)` aka `(x, pattern)`
- Uses function `fixed(pattern)`

'Find' functions: Finding Indices

These are the indices where the pattern match occurs:

```
grep("Rawlings", Sal$Name)
```

```
[1]      9  6854 13284
```

```
which(grepl("Rawlings", Sal$Name))
```

```
[1]      9  6854 13284
```

```
which(str_detect(Sal$Name, "Rawlings"))
```

```
[1]      9  6854 13284
```

'Find' functions: Finding Logicals

These are the indices where the pattern match occurs:

```
head(grepl("Rawlings", Sal$Name))
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
head(str_detect(Sal$Name, "Rawlings"))
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE
```

'Find' functions: finding values, base R

```
grep("Rawlings", Sal$Name, value=TRUE)
```

```
[1] "Rawlings-Blake, Stephanie C" "Rawlings, Kellye A"  
[3] "Rawlings, Paula M"
```

```
Sal[grep("Rawlings", Sal$Name), ]
```

	Name	JobTitle	AgencyID
9	Rawlings-Blake, Stephanie C	MAYOR	A01001
6854	Rawlings, Kellye A	EMERGENCY DISPATCHER	A40302
13284	Rawlings, Paula M	COMMUNITY AIDE	A04015

	Agency	HireDate	AnnualSalary	GrossPay
9	Mayors Office (001)	12/07/1995	167449	\$165249.86
6854	M-R Info Technology (302)	01/06/2003	48940	\$73356.42
13284	R&P-Recreation (015)	12/10/2007	19802	\$10443.70

Showing difference in `str_extract`

`str_extract` extracts just the matched string

```
ss = str_extract(Sal$Name, "Rawling")  
head(ss)
```

```
[1] NA NA NA NA NA NA
```

```
ss[ !is.na(ss) ]
```

```
[1] "Rawling" "Rawling" "Rawling"
```

Showing difference in `str_extract` and `str_extract_all`

`str_extract_all` extracts all the matched strings

```
head(str_extract(Sal$AgencyID, "\\d"))
```

```
[1] "2" "9" "6" "2" "0" "0"
```

```
head(str_extract_all(Sal$AgencyID, "\\d"), 2)
```

```
[[1]]
```

```
[1] "2" "9" "0" "0" "1"
```

```
[[2]]
```

```
[1] "9" "9" "3" "9" "0"
```

Using Regular Expressions

- Look for any name that starts with:
 - Payne at the beginning,
 - Leonard and then an S
 - Spence then capital C

```
head(grep("^Payne.*", x = Sal$Name, value = TRUE), 3)
```

```
[1] "Payne,James R" "Payne,Karen V" "Payne,Jasman T"
```

```
head(grep("Leonard.?S", x = Sal$Name, value = TRUE))
```

```
[1] "Szumlanski,Leonard S" "Payne,Leonard S"
```

```
head(grep("Spence.*C.*", x = Sal$Name, value = TRUE))
```

```
[1] "Spencer,Michael C" "Spencer,Clarence W" "Spencer,Charles A"
```

Using Regular Expressions: **stringr**

```
head(str_subset( Sal$Name, "^Payne.*"), 3)
```

```
[1] "Payne,James R"  "Payne,Karen V"  "Payne,Jasman T"
```

```
head(str_subset( Sal$Name, "Leonard.?S"))
```

```
[1] "Szumlanski,Leonard S" "Payne,Leonard S"
```

```
head(str_subset( Sal$Name, "Spence.*C.*"))
```

```
[1] "Spencer,Michael C"  "Spencer,Clarence W" "Spencer,Charles A"
```


Replace

Let's say we wanted to sort the data set by Annual Salary:

```
class(Sal$AnnualSalary)
```

```
[1] "numeric"
```

```
sort(c("1", "2", "10")) # not sort correctly (order simply ranks the data)
```

```
[1] "1"  "10" "2"
```

```
order(c("1", "2", "10"))
```

```
[1] 1 3 2
```

Replace

So we must change the annual pay into a numeric:

```
head(Sal$AnnualSalary, 4)
```

```
[1] 238772 211785 200000 192500
```

```
head(as.numeric(Sal$AnnualSalary), 4)
```

```
[1] 238772 211785 200000 192500
```

R didn't like the \$ so it thought turned them all to NA.

`sub()` and `gsub()` can do the replacing part in base R.

Replacing and subbing

Now we can replace the \$ with nothing (used `fixed=TRUE` because \$ means ending):

```
Sal$AnnualSalary <- as.numeric(gsub(pattern = "$", replacement="",  
                                   Sal$AnnualSalary, fixed=TRUE))  
Sal <- Sal[order(Sal$AnnualSalary, decreasing=TRUE), ]  
Sal[1:5, c("Name", "AnnualSalary", "JobTitle")]
```

	Name	AnnualSalary	JobTitle
1	Mosby, Marilyn J	238772	STATE'S ATTORNEY
2	Batts, Anthony W	211785	Police Commissioner
3	Wen, Leana	200000	Executive Director III
4	Raymond, Henry J	192500	Executive Director III
5	Swift, Michael	187200	CONTRACT SERV SPEC II

Replacing and subbing: **stringr**

We can do the same thing (with 2 piping operations!) in dplyr

```
dplyr_sal = Sal
dplyr_sal = dplyr_sal %>% mutate(
  AnnualSalary = AnnualSalary %>%
    str_replace(
      fixed("$"),
      "") %>%
    as.numeric() %>%
    arrange(desc(AnnualSalary))
check_Sal = Sal
rownames(check_Sal) = NULL
all.equal(check_Sal, dplyr_sal)
```

```
[1] TRUE
```

Website

[Website](#)