Strings/Characters

What is a string?

A string is a collection of characters. It can be very helpful to be able to manipulate strings to make data usable.

Strings via an Example

Let's consider the Celtics data on my teaching page,

```
data <-
  read.delim("https://.../sportsref_download_5.txt")
```

This data has 3 character variables:

- `Player`, player name
- `Pos`, player position
- `Awards`, player awards

Counting the Number of Characters

The function `nchar` applied to a string counts the number of characters.

For example,

```
nchar("Celtics")
```

```
[1] 7
```

When applied to a vector of strings, `nchar` counts the number of characters per element of the vector.

```
nchar(data$Pos)
```

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

Subsetting a String By Character Position

Sometimes we might want to take a substring by extracting the characters in certain positions, e.g. we may want to extract just the first character.

```
substr(data$Pos, 1, 1)
```

```
 [1] "P" "S" "S" "P" "P" "S" "C" "C" "C" "S" "S" "C" "P" "P
```

The first number tells `substr` the position of the first character to extract, and the last number tells `substr` the position of the last character.

```
substr(data$Pos, 2, 2)
```

```
 [1] "F" "G" "F" "G" "G" "F" ""  ""  ""  "F" "F" ""  "F" "C
```

Note - if a string only has one character, then extracting the second character returns nothing, "".

Subsetting a String By Character Position

You can pass **vectors** of starting and ending positions to `substr`. For instance, suppose you wanted to extract the last letter of variable `Player` for each player.

How would you do it?

Subsetting a String By Character Position

We can create new variables by subsetting, e.g. we might want to make a new variable based on `Pos` that just indicates whether a player is a center, a forward, or a guard.

```
data$CGF <- ifelse(nchar(data$Pos) == 1, data$Pos,
                   substr(data$Pos, 2, 2))
```

```
table(data$CGF)
```

```
C F G
4 9 6
```

Splitting Strings

Suppose we want to break a string apart. For instance, we may want to do this if we want to create separate variables for first and last name.

The `strsplit` function takes a vector of strings and splits them wherever the string indicated by the `split` argument appears.

```
names <- strsplit(data$Player, split = " ")
```

It returns a **list** with the same length as the vector of strings.

Each element of the list has a variable number of elements, depending on how the string used to split appeared.

What strsplit Returns

```
names[[1]]
```

```
[1] "Jayson" "Tatum"
```

```
names[[13]]
```

```
[1] "Xavier"  "Tillman" "Sr."
```

We will want to manipulate the output of `strsplit` using the functions we learned about for working with lists, `lapply` and `unlist`.

Summarizing Strings Created by `strsplit`

We can summarize the number of substrings created by splitting player names whereever a space appears.

```
table(unlist(lapply(names, length)))
```

```
 2  3
18  1
```

Extracting Individual Substrings

Since the player's first name is always the first substring returned by splitting on spaces, we can create a new variable that corresponds to the first substring.

```
data$First <-
  unlist(
    lapply(
      strsplit(data$Player, " "),
      function(x) {x[1]}))
```

Extracting and Combining Individual Substrings

Some players have last names that include a space, which leads to last names being made up of multiple substrings created by splitting on spaces.

This means that creating a last name variable will require learning how to **combine** strings.

Combining Strings

The `paste` function allows us to combine strings.

It takes a collection of strings and an argument called `sep`, which describes how the strings are separated when combined.

```
paste("Boston", "Celtics")
```

```
[1] "Boston Celtics"
```

The default when `sep` is not specified is to separate strings with a single space.

```
paste("Boston", "Celtics", sep = " ")
```

```
[1] "Boston Celtics"
```

```
paste("Boston", "Celtics", sep = "")
```

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

Combining Strings

The `paste` function can also be applied to vectors, in multiple ways.

If multiple vectors are specified, it will paste them together element by element and return a vector that is the same length as the longest vector that was supplied.

```
paste(c("Boston", "Los Angeles"), c("Celtics", "Lakers"))
```

```
[1] "Boston Celtics"     "Los Angeles Lakers"
```

Does paste "Recycle"?

Yes. Be careful!

```
paste(c("Boston", "Los Angeles"), c("Celtics"))
```

```
[1] "Boston Celtics"      "Los Angeles Celtics"
```

It returns a vector that is the same length as the longest vector that was supplied.

```
paste(c("Boston"), c("Celtics", "Lakers"))
```

```
[1] "Boston Celtics" "Boston Lakers"
```

Combining Elements of a Vector

Sometimes, we may want to use `paste` to combine all elements of a single vector. To do this, we need to specify the `collapse` argument instead of the `sep` argument.

```
paste(c("Boston", "Celtics"), collapse = " ")
```

```
[1] "Boston Celtics"
```

The string provided to `collapse` will be used to separate elements of the provided vector when they are combined.

Creating a Last Name Variable

We can use what we just learned to create a last name variable!

```
data$Last <- unlist(
  lapply(strsplit(data$Player, " "),
         function(x) {
           paste(x[2:length(x)], collapse = " ")}))
```

Creating Indicators for Awards

We can use what we've learned to create indicator variables for:

- Defensive Player of the Year
- All Star

An indicator or dummy variable takes on value of 1 or 0 depending on whether or not a statement is true.

Try it!

Searching Strings

There are two useful functions for searching strings:

- `grep`, which takes string to search for, a vector of strings to search in, and returns a vector of indices for which the searched string appears
- `grepl`, which takes string to search for, a vector of strings to search in, and returns a logical vector of that is `TRUE` when the searched string appears

```
grep("Ja", data$First)
```

```
[1]  1  3 15
```

```
grepl("Ja", data$First)
```

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

Creating Indicators for Awards with `grepl`

You can use `grepl` to create Defensive Player of the Year and All Star indicators.

Try it!

Modifying Strings with `sub` and `gsub`

There are two useful functions for modifying strings via substitution:

- `sub` takes a string to look for, a new string to replace the string to look for with (once), and a vector of strings to apply the replacement to
- `gsub` takes a string to look for, a new string to replace the string to look for with (as many times as needed), and a vector of strings to apply the replacement to

Demonstration of sub and gsub

```
data$Awards[1:4]
```

```
[1] "MVP-6,CPOY-9,AS,NBA1" "DPOY-8"                "AS"
[4] "DPOY-6"
```

```
sub("-", "", data$Awards[1:4])
```

```
[1] "MVP6,CPOY-9,AS,NBA1" "DPOY8"                 "AS"
[4] "DPOY6"
```

```
gsub("-", "", data$Awards[1:4])
```

```
[1] "MVP6,CPOY9,AS,NBA1" "DPOY8"                 "AS"
[4] "DPOY6"
```

Creating `CGF` Variable with `sub` or `gsub`

We can remake the new variable based on `Pos` that just indicates whether a player is a center, a forward, or a guard more easily with `sub` or `gsub`.

Try it!

Finding Starting Position

There are two useful functions for finding the position where a specified string begins:

- `regexpr`, takes a string to look for (once), and a vector of strings to look in
- `gregexpr`, takes a string to look for (multiple times), and a vector of strings to look in

Finding Starting Position with `regexpr`

```
data$Player[1]
```

```
[1] "Jayson Tatum"
```

```
regexpr("a", data$Player[1])
```

```
[1] 2
attr(,"match.length")
[1] 1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

Finding Starting Position with regexpr

```
data$Player[1]
```

```
[1] "Jayson Tatum"
```

```
gregexpr("a", data$Player[1])
```

```
[[1]]
[1] 2 9
attr(,"match.length")
[1] 1 1
attr(,"index.type")
[1] "chars"
attr(,"useBytes")
[1] TRUE
```

Finding Starting Position for Vectors

```
regexpr("a", data$Player)
```

```
gregexpr("a", data$Player)
```

Regular Expressions

Remember when we saw all of the options for the `LIKE` command in SQL, which allows us to find make more specific requests for text matches?

Regular expressions are a related concept.

They allow us to look for broad types of patterns in strings.

Using Regular Expressions

We can put letters in brackets to look for strings that contain any of the letters

```
grep("[ao]", data$First)
```

```
 [1]  1  3  5  6  8 10 12 13 14 15 16 17
```

We can also search for strings of a specific length that have arbitrary letters where ever a period appears

```
grep("Jay..n", data$First)
```

```
[1] 1 3
```

Special Characters

Note - we just saw that when using regular expression functions e.g. `grep`, certain characters such as `[` or `.` have a special meaning.

```
grep(".", data$Last)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
```

To actually look for these characters, we need to either specify an argument `fixed = TRUE` **or** use escapes characters `\\` to tell `R` to look for the specific character

```
grep(".", data$Last, fixed = TRUE)
```

```
[1] 13
```

```
grep("\\.", data$Last)
```

```
[1] 13
```

Reading in Text Line by Line

We can read in a text file line by line using the `readLines` command.

This creates a vector with one element per line.

For instance, we could read any of our `.csv` or `.txt` files in this way.

```
lines <-
  readLines("https://.../sportsref_download_5.txt")
```

Reading in Text Line by Line

You may frequently get the following warning when using `readLines`.

```
Warning in
readLines("https://maryclare.github.io/content/courses/stat
incomplete final line found on
'https://maryclare.github.io/content/courses/statisticalcon
```

This means that the last line didn't technically end with a line break.

`R` doesn't like that, but everything will still work ok.

If you want to make the error message go away you can open the file in your preferred text editor and add a line break at the end, so that the file ends with an empty line.

What `readLines` Provides

We can look at a few lines of output from `readLines` to get a sense of what it provides:

```
lines[1]
```

```
[1] "Rk\tPlayer\tAge\tPos\tG\tGS\tMP\tFG\tFGA\tFG%\t3P\t3PA
```

```
lines[2]
```

```
[1] "1\tJayson Tatum\t25\tPF\t74\t74\t2645\t9.1\t19.4\t0.47
```

Reading in Data from HTML Source Code

As an activity, we're going to use `readLines` and what we have learned about working with strings to read in the same Celtics data we've been using from the source code:

https://www.basketball-reference.com/teams/BOS/2024.html#all_per_minute_stats