Week 8 Module 1: String Functions

CSCI E-5a: Programming in R

Let's clear the global computing environment:

```
rm( list = ls() )
```

Now we can load in the Module 1 R objects:

```
load( "Module 1 R Objects.Rdata" )
```

Let's list the objects that we just loaded in:

```
ls()
```

```
## [1] "exercise.1.data"
```

Module Overview and Learning Outcomes

Hello! And welcome to Module 1: String Functions.

In this module, we'll explore techniques for working with character strings.

- In section 1, we'll meet some basic utility functions for character strings.
- In section 2, we'll use our utility string functions to strip out internal codes from strings.
- In section 3, we'll use our utility string functions to correctly format baseball batting statistics using standard formatting conventions.
- In section 4, we'll meet the powerful `grep()` and `grepl()` functions, which enable us to perform substring pattern matching.

Once you've finished this module, you'll be able to:

- Perform operations on strings using the utility string functions.
- Use the utility functions to strip out internal codes from identifier strings.
- Use the utility functions to properly format baseball batting statistics using standard formatting conventions.
- Use the `grep()` and `grepl()` functions to perform substring pattern matching.

There are 5 new built-in R functions in this module:

- `paste()`
- `nchar()`
- `substr()`
- `grep()`
- `grepl()`

All right! Let's get started by learning about some basic utility functions for working with character string values.

Section 1: String Utility Functions

> **Main Idea:** *We can use utility functions to work with character strings*

In this section, we'll meet some basic utility functions for character strings.

Let's look at some functions that will provide a lot of value to you in a small package:

- The `paste()` function
- The `nchar()` function
- The `substr()` function

The `paste()` function

The `paste()` function takes multiple character strings and combines them together into one character string.

```
paste( "Hi", "Mom", "!" )
```

```
## [1] "Hi Mom !"
```

Notice here that by default `paste()` inserts a space as a separation character; you can adjust this using the `sep =` option.

It's important to understand the differences between the `paste()` function and the `c()` function.

Look at what the `c()` function does with the same three inputs:

```
c( "Hi", "Mom", "!" )
```

```
## [1] "Hi"  "Mom" "!"
```

Do you see the difference?

With the `c()` function, the 3 inputs are combined into a `character` vector with 3 elements.

With the `paste()` function, the three inputs are concatenated into a single string.

That is, the output of the `c()` function is a vector with three elements, while the output of the `paste()` function is just a single character string.

It's also important to understand the difference between the `paste()` function and the `formatC()` function.

The `formatC()` function takes a numeric value as an input argument, and returns a character string consisting of a formatted representation of the input numeric value.

The `paste()` function doesn't do *any* formatting at all.

If you use a numeric value as one of the input values to the `paste()` function, you can end up with a lot of digits:

```
paste( "pi =", pi )
```

```
## [1] "pi = 3.14159265358979"
```

This is actually more digits than if we directly display the value:

```
pi
```

```
## [1] 3.141593
```

By itself, the `paste()` function can't do any formatting of numerical values.

However, we can use the `formatC()` function to do the formatting, and `paste()` to assemble the final string:

```
paste(
    "pi =",
    formatC(
        x = pi,
        format = "f",
        digits = 2
    )
)
```

```
## [1] "pi = 3.14"
```

Notice the quotes around the return value from this function.

That's because the `paste()` function returns a character string.

This is an R object, and we can store it in a variable:

```
pi.string <-
    paste(
        "pi =",
        formatC(
            pi,
            format = "f",
            digits = 2
        )
    )
```

Now that we've stored this string in a variable, we can recall it:

```
pi.string
```

```
## [1] "pi = 3.14"
```

So the strategy here is: use a function such as `formatC()` to do any fancy formatting, and then use the `paste()` function to assemble a collection of strings into one composite string.

The `nchar()` function

The `nchar()` function takes a character string and returns the number of characters in the string:

```
nchar( "Hedgehog" )
```

```
## [1] 8
```

I realize this doesn't seem very exciting, but we'll see that it can be very helpful as a utility function.

The `substr()` function

The `substr()` function enables us to extract a substring from a character string.

The `substr()` function takes three input arguments:

- The first input argument, denoted `x`, is a character string.
- The second input argument, denoted `start`, indicates the location in the first input argument where the substring starts.
- The third input argument, denoted `stop`, indicates the location in the first input argument where the substring stops.

Then the `substr()` function returns the substring of the first input argument, starting at the location specified by `start` and ending at the location specificed by `stop`.

The character at the ending index will be included in the substring.

Here's an example of `substr()` at work.

We'll create a string to hold the value "Kind of Blue", and then we'll extract the substring starting at character position 9 and ending at character position 12:

```
album.title <- "Kind of Blue"

substr( album.title, start = 9, stop = 12 )
```

```
## [1] "Blue"
```

When I write the word "extract", that does not imply that the original variable is in any way modified:

```
album.title
```

```
## [1] "Kind of Blue"
```

So those are some basic utility functions for character strings.

Now let's see how to use these tools to solve a real-world problem.

Section 2: Stripping Out Internal Codes

> **Main Idea:** *We can strip out internal codes from strings*

In this section, we'll use our utility string functions to strip out internal codes from strings.

Let's take a look at how we can use these string functions to solve a small but important real-world problem.

We are given a vector of character strings, where each string consists of two parts:

- One part is some form of an internal code, which we are not interested in.
- The second part is an identifier, which we *are* interested in.

Our goal in the problem is to filter out the internal codes, and obtain a vector that consists of only the identifier component of each string.

There are a number of variations on this problem, depending on which comes first, either the internal code or the identifier, and whether or not these have a fixed or variable length.

We'll start by considering the simplest version of this problem, in which we have a fixed length for both the internal code and the identification number.

In this example, the first 10 characters of the string are the internal identification code that we want to strip out, and the last five characters of the string are the customer identification numbers that we want to keep.

For instance, the strings might look like this:

Full String	First Part	Second Part
"XYZ-4-ABC_45613"	"XYZ-4-ABC_"	"45613"
"XYZ-4-ABC_71634"	"XYZ-4-ABC_"	"71634"
"XYZ-4-ABC_59732"	"XYZ-4-ABC_"	"59732"
"XYZ-4-ABC_60841"	"XYZ-4-ABC_"	"60841"
"XYZ-4-ABC_39782"	"XYZ-4-ABC_"	"39782"

Let's start with the very first string.

Do you see how to extract the last 5 characters?

We can use the `substr()` function:

```
substr( "XYZ-4-ABC_45613", start = 11, stop = 15 )
```

```
## [1] "45613"
```

Now we can do this with the second string:

```
substr( "XYZ-4-ABC_71634", start = 11, stop = 15 )
```

```
## [1] "71634"
```

In fact, we can vectorize this process.

Let's make a vector of these strings:

```
version.1.vector <-
    c(
      "XYZ-4-ABC_45613",
      "XYZ-4-ABC_71634",
      "XYZ-4-ABC_59732",
      "XYZ-4-ABC_60841",
      "XYZ-4-ABC_39782"
    )
```

Now can you see how to do this for the entire vector?

We can use a vectorized operation:

```
substr( version.1.vector, start = 11, stop = 15 )
```

```
## [1] "45613" "71634" "59732" "60841" "39782"
```

Just what we wanted!

Now let's consider a slightly more complicated scenario.

In this example, once again the first part of the character string is an internal code with the form "XYZ-4-ABC_" and the second part of the character string is an identification number, just as in our previous example.

However, now the identification number can have a variable length.

For instance, the strings might look like this:

Full String	First Part	Second Part
"XYZ-4-ABC_1452"	"XYZ-4-ABC_"	"1452"
"XYZ-4-ABC_632"	"XYZ-4-ABC_"	"632"
"XYZ-4-ABC_71984"	"XYZ-4-ABC_"	"71984"
"XYZ-4-ABC_5233"	"XYZ-4-ABC_"	"5233"
"XYZ-4-ABC_0837518"	"XYZ-4-ABC_"	"0837518"

Our goal is to strip off the first part, and end up with a sequence which consists of only the second parts of each string.

Notice that the first part that we want to strip out has a fixed length, but the second part with the identifiers has a variable length.

That means that we can't simply use a constant value for the endpoint of the identifier string, because it has a variable length and thus the endpoint is not a fixed value.

However, we can use the `nchar()` function to find the endpoint.

For instance, with the first string, we have:

```
substr(
  x = "XYZ-4-ABC_1452",
  start = 11,
  stop = nchar( "XYZ-4-ABC_1452" )
)
```

```
## [1] "1452"
```

Now we can do this with the second string:

```
substr(
  x = "XYZ-4-ABC_632",
  start = 11,
  stop = nchar( "XYZ-4-ABC_632" )
)
```

```
## [1] "632"
```

Here's a vector version of this:

```
version.2.vector <-
    c(
      "XYZ-4-ABC_1452",
      "XYZ-4-ABC_632",
      "XYZ-4-ABC_71984",
      "XYZ-4-ABC_5233",
      "XYZ-4-ABC_0837518"
    )
```

Now we can map this code across the entire vector:

```
substr(
    version.2.vector,
    start = 11,
    stop = nchar( version.2.vector )
)
```

```
## [1] "1452"    "632"     "71984"   "5233"    "0837518"
```

Incidentally, the inspiration for this entire section was a problem that a student brought to me one time after class that was essentially this version of stripping out internal codes.

We can make this even more challenging.

Now, we still want to strip out the first part, but it can have variable length.

The second part consisting of the identifiers has a fixed length, though.

Full String	First Part	Second Part
"XYZ-4-ABC_14572"	"XYZ-4-ABC_"	"14572"
"XYZ-613-ABC_63250"	"XYZ-613-ABC_"	"63250"

Full String	First Part	Second Part
“XYZ-79-ABC_71984”	“XYZ-79-ABC_”	“71984”
“XYZ-4698-ABC_52133”	“XYZ-4698-ABC_”	“52133”
“XYZ-71275-ABC_08375”	“XYZ-71275-ABC_”	“08375”

In this case, each string can have a different length, and the starting point of the identifiers will be different for each string.

However, we know that the customer identifiers all have the same length, so we can work backwards from the end, which we can calculate using `nchar()`.

We can extract the identifier from the first string by using the `nchar()` function twice:

```
substr(
  x = "XYZ-4-ABC_14572",
  start = nchar( "XYZ-4-ABC_14572" ) - 4,
  stop = nchar( "XYZ-4-ABC_14572" )
)
```

```
## [1] "14572"
```

As before, we can vectorize this process.

Let’s create a vector to hold the character strings for this version:

```
version.3.vector <-
    c(
      "XYZ-4-ABC_14572",
      "XYZ-613-ABC_63250",
      "XYZ-79-ABC_71984",
      "XYZ-4698-ABC_52133",
      "XYZ-71275-ABC_08375"
    )
```

And now we can vectorize this operation:

```
substr(
  x = version.3.vector,
  start = nchar( version.3.vector ) - 4,
  stop = nchar( version.3.vector )
)
```

```
## [1] "14572" "63250" "71984" "52133" "08375"
```

So that how to use our utility string functions to strip out internal codes from strings.

Now let’s see how to use these utility string functions to properly format baseball batting statistics.

Exercise 1: Stripping out an internal code

The vector `exercise.1.data` consists of character strings, each of which begins with an internal code of fixed length, followed by an identifier of variable length consisting entirely of digits.

For each character string value in `exercise.1.data`, strip out the internal code of fixed length, and return a character string vector consisting of just the variable length identifiers.

You'll have to figure out how long the internal code by examining the data yourself.

Solution

Section 3: Baseball Batting Statistics

> **Main Idea:** *We can use string functions to properly format baseball batting statistics*

In this section, we'll use our utility string functions to correctly format baseball batting statistics using standard formatting conventions.

Let's return to an unresolved issue from Lecture 2, concerning the proper display of baseball batting statistics.

Whenever we've reported baseball batting statistics, we've always used three decimal places.

In fact, I've explicitly told you that that was the standard, and anything else would be considered "weird".

To achieve this, we've used the `formatC()` function to control the formatting so that we always display batting averages with exactly 3 decimal places.

For instance, Babe Ruth had 2,873 hits in 8,399 at-bats, so his career batting average was 0.342:

```
babe.ruth.at.bats <- 8399

babe.ruth.hits <- 2873

babe.ruth.career.batting.average <-
  babe.ruth.hits / babe.ruth.at.bats

cat(
    "Babe Ruth career batting average:",
    formatC(
        x = babe.ruth.career.batting.average,
        format = "f",
        digits = 3
    )
)
```

```
## Babe Ruth career batting average: 0.342
```

But in fact this *still* isn't right.

The problem isn't the number of digits to the right of the decimal place – it's the number of digits to the left.

Traditionally, baseball statistics are reported without the leading 0, so that Babe Ruth's batting average is written as ".342".

How can we write R code that will print out a baseball player's batting average in the correct format i.e. with no leading zero, and three digits to the right of the decimal point?

If you can, think about this for a moment.

We know that we need to use the `formatC()` function to be sure that exactly 3 decimal places will be displayed.

To get rid of the leading '0' character, we can use the `substr()` function in conjunction with the `formatC()` function.

First, we can use the `formatC()` function to format the batting average so that it has a leading 0 before the decimal point and 3 digits after the decimal point.

Thus, including the decimal point, we know that this formatted string must have exactly 5 characters:

```
babe.ruth.career.batting.average.string <-
  formatC(
    x = babe.ruth.career.batting.average,
    format = "f",
    digits = 3
  )

babe.ruth.career.batting.average.string
```

```
## [1] "0.342"
```

Now we can use the `substr()` function to strip out the leading 0:

```
babe.ruth.career.batting.average.formatted.string <-
  substr(
    x = babe.ruth.career.batting.average,
    start = 2,
    stop = 5
  )

babe.ruth.career.batting.average.formatted.string
```

```
## [1] ".342"
```

Let's put this all together, and then report the result by using a `cat()` statement:

```
babe.ruth.at.bats <- 8399

babe.ruth.hits <- 2873

babe.ruth.career.batting.average <-
  babe.ruth.hits / babe.ruth.at.bats

babe.ruth.career.batting.average.string <-
  formatC(
    x = babe.ruth.career.batting.average,
    format = "f",
    digits = 3
  )

babe.ruth.career.batting.average.formatted.string <-
  substr(
    x = babe.ruth.career.batting.average,
    start = 2,
    stop = 5
```

```
  )

cat(
    "Babe Ruth career batting average:",
    formatC(
        x = babe.ruth.career.batting.average.formatted.string,
        format = "f",
        digits = 3
    )
)
```

```
## Babe Ruth career batting average: .342
```

So that's how to use our utility string functions to correctly format baseball batting statistics using standard formatting conventions.

Now let's learn about the amazing `grep()` and `grepl()` functions.

Exercise 2: Ted Williams' On-Base Percentage

Here are Ted Williams's career statistics:

Statistics	Value
Plate appearances	9,792
At-bats	7,706
Hits	2,654
Doubles	525
Triples	71
Home Runs	521
Bases on balls	2,021
Hit by a pitch	39
Sacrifice flies	20

Report Ted Williams' On-Base Percentage (OBP) using a `cat()` statement, with the value displayed using the standard baseball conventions.

Solution

Section 4: The `grep()` and `grepl()` functions

> **Main Idea:** *We can use the `grep()` and `grepl()` functions to perform sophisticated text pattern matching*

In this section, we'll meet the powerful `grep()` and `grepl()` functions, which enable us to perform substring pattern matching.

The `grep()` and `grepl()` functions can perform substring pattern matching on a character string.

They are somewhat specialized, but are also extremely powerful.

In fact, there's a whole family of these specialized pattern-matching functions, but I'm only going to show you `grep()` and `grepl()`.

The `grep()` function takes two input arguments:

- A character string that represents some sort of text pattern, denoted `pattern`.
- A vector of character strings as input data, denoted `x`.

The `grep()` function then attempts to find a match between the pattern and each of the character strings in the `x` input data vector.

The pattern doesn't have to match any of the character strings *exactly*; instead, it only has to match a substring of a character string.

This can be very useful when we are working with loosely defined data that can arise when data are entered by hand.

For instance, suppose a medical researcher has a sequence of patient treatments like this:

Patient	Treatment
1	"insulin"
2	"no medications"
3	"anti-coagulant and insulin"
4	"anti-depressant"
5	"no medications"
6	"some insuline"

The researcher would like to search for patients who are on insulin.

For instance, let's suppose we have a vector of character strings like this:

```
medication.character.string.vector <-
  c(
    "insulin",
    "no medications",
    "anti-coagulant and insulin",
    "anti-depressant",
    "no medications",
    "some insuline"
  )
```

We want to search for strings that match the pattern "insulin".

Notice that there is only one element in the sequence that is exactly equal to the pattern "insulin", but there are two others that contain the pattern as a substring.

So we can't just search for exact equality between the pattern and each character string.

Instead, the `grep()` function will perform substring pattern matching for the pattern string with each of the elements in the input data vector `x`:

```
grep(
  pattern = "insulin",
  x = medication.character.string.vector
)
```

```
## [1] 1 3 6
```

Then `grep()` returns a vector consisting of the locations of the elements of `medication.character.string.vector` that have a substring that matches the pattern "insulin".

We can save the output of the `grep()` function in a vector, and then use this to select the elements of the vector that contain a substring match with the pattern:

```
pattern.match.index.vector <-
  grep(
    pattern = "insulin",
    x = medication.character.string.vector
  )

medication.character.string.vector[
  pattern.match.index.vector
]
```

```
## [1] "insulin"                    "anti-coagulant and insulin"
## [3] "some insuline"
```

A similar function is `grepl()`, which returns a logical vector consisting of `TRUE` and `FALSE` values that are `TRUE` precisely when the corresponding element of the `x` vector has a substring that matches the pattern string.

```
grepl(
  pattern = "insulin",
  x = medication.character.string.vector
)
```

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

Here the `grepl()` function is returning a vector with `TRUE` values in positions 1, 3, and 6, which are the positions in the character string vector that have a substring pattern match.

The `grepl()` function works very nicely with the `ifelse()` function.

For instance, we can create a new vector consisting of either "insulin" or "no insulin" for this medication data:

```
insulin.vector <-
  ifelse(
    test =
      grepl(
        pattern = "insulin",
        x = medication.character.string.vector
      ),
    yes = "insulin",
    no = "no insulin"
  )

insulin.vector
```

```
## [1] "insulin"    "no insulin" "insulin"    "no insulin" "no insulin"
## [6] "insulin"
```

Notice that the last element of the input vector was "some insuline", which is misspelled.

Nonetheless, the `grep()` function was able to match the pattern to a substring.

If you have data that is somewhat irregular, a little bit of ingenuity along with the `grep()` function can often help you to clean your data efficiently.

For instance, suppose we want to create a new vector of medications, but where all the elements that match the pattern "insulin" in the original data are replaced with the exact character string "insulin":

```
cleaned.medication.character.string.vector <-
  ifelse(
    test =
      grepl(
        pattern = "insulin",
        x = medication.character.string.vector
      ),
    yes = "insulin",
    no = medication.character.string.vector
  )

cleaned.medication.character.string.vector
```

```
## [1] "insulin"         "no medications"  "insulin"         "anti-depressant"
## [5] "no medications"  "insulin"
```

```
insulin.locations.vector <-
    grep(
        pattern = "insulin",
        x = medication.character.string.vector
    )

insulin.vector <-
    medication.character.string.vector

insulin.vector[ insulin.locations.vector ] <- "insulin"

insulin.vector
```

```
## [1] "insulin"         "no medications"  "insulin"         "anti-depressant"
## [5] "no medications"  "insulin"
```

By repeated application of this process, you can handle some very difficult cases.

> **Practical tip:** *When you're cleaning data, you don't have to do it all in just one step.*

By the way, the `grep()` and `grepl()` functions are case-sensitive, which means that they require the exact upper- and lower-case letters for a match.

For instance, consider this variant of the original `medication.character.string.vector`, in which the first element is replaced with a string with all capital letters:

```
medication.character.string.2.vector <-
  c(
    "INSULIN",
    "no medications",
    "anti-coagulant and insulin",
    "anti-depressant",
    "no medications",
    "some insuline"
  )
```

When we use the `grep()` or `grepl()` functions with this new data, they will not match the pattern string "insulin" to the first element of the input data, because by default these functions are case-sensitive:

```
grep(
  pattern = "insulin",
  x = medication.character.string.2.vector
)
```

```
## [1] 3 6
```

```
medication.character.string.2.vector
```

```
## [1] "INSULIN"                    "no medications"
## [3] "anti-coagulant and insulin" "anti-depressant"
## [5] "no medications"             "some insuline"
```

However, we can set the very helpfully named option `ignore.case` to `TRUE`, and the `grep()` will use case-insensitive matching:

```
grep(
  pattern = "insulin",
  x = medication.character.string.2.vector,
  ignore.case = TRUE
)
```

```
## [1] 1 3 6
```

The `grep()` function might seem a little mysterious at first, but with a little practice you should feel comfortable with it.

The `grep()` function is very deep, and if you are interested there's an excellent discussion on Wikipedia.

In fact, there is an entire family of functions related to `grep()` and `grepl()`, and we've only scratched the surface here.

Many people will not need to do this sort of substring pattern matching, so I don't want to emphasize `grep()` and `grepl()` too much.

However, if you *do* need to use these tools, they can be very helpful.

One area where this sort of pattern-matching is very useful is bioinformatics.

There's a lot more to learn about this topic, so if you are interested you'll have to go out and do some research on your own.

So that's how to use the `grep()` and `grepl()` functions for character string pattern matching.

Now let's review what we've learned in this module.

Exercise 3: Substring pattern matching with the grep() function

Consider this set of data:

```
exercise.3.data <-
  c(
    "healthy kale",
    "Sugar Bms",
    "Sugar Bom",
    "Krispy Ym!",
    "kale Tfuo",
    "Sug Bz",
    "Crispy Yum",
    "healthy kale and tofu"
  )
```

Part (a): Counting rows with "kale"

How many rows in `exercise.3.data` contain the string "kale"?

Solution

Part (b): Converting to "kale" or "no kale"

Create a new vector consisting of the values "kale" or "no kale", where this new vector has the value "kale" precisely when the input data vector `exercise.3.data` has an element with a substring that matches the pattern string "kale".

Solution

Part (c): Converting all "kale" entries to "HKT"

Create a new vector consisting of the values of the original data vector `exercise.3.data`, but with all elements that match the pattern "kale" replaced with "HKT".

Solution

Module Review

In this module, we explored techniques for working with character strings.

- In section 1, we met some basic utility functions for character strings.
- In section 2, we used our utility string functions to strip out internal codes from strings.
- In section 3, we used our utility string functions to correctly format baseball batting statistics using standard formatting conventions.
- In section 4, we met the powerful `grep()` and `grepl()` functions, which enable us to perform substring pattern matching.

Now that you've finished this module, you should be able to:

- Perform operations on strings using the utility string functions.
- Use the utility functions to strip out internal codes from identifier strings.
- Use the utility functions to properly format baseball batting statistics using standard formatting conventions.
- Use the `grep()` and `grepl()` functions to perform substring pattern matching.

There were 5 new built-in R functions in this module:

- `paste()`
- `nchar()`
- `substr()`
- `grep()`
- `grepl()`

All right! That's it for Module 1: String Functions.

Now let's move on to Module 2: R Functions.

Solution to the Exercises

Exercise 1: Stripping out an internal code

The vector `exercise.1.data` consists of character strings, each of which begins with an internal code of fixed length, followed by an identifier of variable length consisting entirely of digits.

For each character string value in `exercise.1.data`, strip out the internal code of fixed length, and return a character string vector consisting of just the variable length identifiers.

You'll have to figure out how long the internal code by examining the data yourself.

Solution

Let's first take a look at the values in `exercise.1.data` by directly displaying the vector:

```
exercise.1.data
```

```
## [1] "pfW8Cs.5970"    "pfW8Cs.0812"    "pfW8Cs.2571"    "pfW8Cs.679058"
## [5] "pfW8Cs.3508427" "pfW8Cs.758162"  "pfW8Cs.2068471" "pfW8Cs.3491852"
```

You can see that the first 7 characters of each string are always the same, so this is the internal code.

Then we can extract the identifier string by using the `substr()` function and the `nchar()` function:

```
identifier.string.vector <-
  substr(
    x = exercise.1.data,
    start = 8,
    stop = nchar( exercise.1.data )
  )
```

Now let's display this directly:

```
identifier.string.vector
```

```
## [1] "5970"    "0812"    "2571"    "679058"  "3508427" "758162"  "2068471"
## [8] "3491852"
```

Exercise 2: Ted Williams' On-Base Percentage

Here are Ted Williams's career statistics:

Statistics	Value
Plate appearances	9,792
At-bats	7,706
Hits	2,654
Doubles	525
Triples	71
Home Runs	521
Bases on balls	2,021
Hit by a pitch	39
Sacrifice flies	20

Report Ted Williams' On-Base Percentage (OBP) using a `cat()` statement, with the value displayed using the standard baseball conventions.

Solution

Let's start by creating a set of variables to hold Williams' career batting statistics:

```
ted.williams.at.bats <- 7706

ted.williams.hits <- 2654

ted.williams.doubles <- 525

ted.williams.triples <- 71

ted.williams.home.runs <- 521

ted.williams.bases.on.balls <- 2021

ted.williams.hit.by.a.pitch <- 39

ted.williams.sacrifice.flies <- 20
```

Now we can calculate Ted Williams' career on-base percentage:

```
ted.williams.on.base.percentage <-
    (ted.williams.hits +
         ted.williams.bases.on.balls +
         ted.williams.hit.by.a.pitch) /
    (ted.williams.at.bats +
         ted.williams.bases.on.balls +
         ted.williams.hit.by.a.pitch +
         ted.williams.sacrifice.flies)
```

Next, we'll format this result using the `formatC()` function so that it has exactly 3 digits trailing the decimal point, and then directly display it to check our work:

```
ted.williams.on.base.percentage.string <-
  formatC(
    x = ted.williams.on.base.percentage,
    format = "f",
    digits = 3
  )

ted.williams.on.base.percentage.string
```

```
## [1] "0.482"
```

Now let's use the `substr()` function to remove the leading 0, and then directly display this string to check our work:

```
ted.williams.on.base.percentage.formatted.string <-
  substr(
    x = ted.williams.on.base.percentage.string,
    start = 2,
    stop = 5
  )

ted.williams.on.base.percentage.formatted.string
```

```
## [1] ".482"
```

Finally, we can report this with a `cat()` statement:

```
cat(
  "Ted Williams on-base percentage:",
  ted.williams.on.base.percentage.formatted.string
)
```

```
## Ted Williams on-base percentage: .482
```

Exercise 3: Substring pattern matching with the grep() function

Consider this set of data:

```
exercise.3.data <-
  c(
    "healthy kale",
    "Sugar Bms",
    "Sugar Bom",
    "Krispy Ym!",
    "kale Tfuo",
    "Sug Bz",
    "Crispy Yum",
    "healthy kale and tofu"
  )
```

Part (a): Counting rows with "kale"

How many rows in `exercise.3.data` contain the string "kale"?

Solution

```
kale.match.index.vector <-
  grep(
    pattern = "kale",
    x = exercise.3.data
  )

length( kale.match.index.vector )
```

```
## [1] 3
```

Part (b): Converting to "kale" or "no kale"

Create a new vector consisting of the values "kale" or "no kale", where this new vector has the value "kale" precisely when the input data vector `exercise.3.data` has an element with a substring that matches the pattern string "kale".

Solution

```
kale.match.logical.indexing.vector <-
  grepl(
    pattern = "kale",
    x = exercise.3.data
  )

ifelse(
  test = kale.match.logical.indexing.vector,
  yes = "kale",
  no = "no kale"
)
```

```
## [1] "kale"    "no kale" "no kale" "no kale" "kale"    "no kale" "no kale"
## [8] "kale"
```

Part (c): Converting all "kale" entries to "HKT"

Create a new vector consisting of the values of the original data vector `exercise.3.data`, but with all elements that match the pattern "kale" replaced with "HKT".

Solution

```
kale.match.logical.indexing.vector <-
  grepl(
    pattern = "kale",
    x = exercise.3.data
  )

ifelse(
```

```
  test = kale.match.logical.indexing.vector,
  yes = "HKT",
  no = exercise.3.data
)
```

```
## [1] "HKT"        "Sugar Bms"  "Sugar Bom"  "Krispy Ym!" "HKT"
## [6] "Sug Bz"     "Crispy Yum" "HKT"
```