Summary Sheet

AUTHOR
Julia Gallucci

Class 2

Main components of a "Reprex" (reproducible example)

1. **Environment:** calls for any necessary libraries and information about your R environment that might be relevant

   session

   ```
   sessionInfo() #to get version information about R, the OS and attached or loaded packages.

   R.Version()$version.string #provides detailed information about the version of R running.

   RStudio.Version()``$version #provides detailed information about the version of RStudio running.
   ```

2. **Toy data set:** a minimal data set that the code can be run on. i.e., if you have a large data set, you can select a subset of it and attach that with your reprex.

3. **Code:** minimal and runnable code that recreates the error

   ```
   library(reprex)

   reprex({

   #code that is producing the error

   })
   ```

Best coding practices:

- Well-comment your code (`using #`)
- Name your variables so they are meaningful and descriptive (`i.e., avoid using x)`
  - To separate words when naming your variable use camelCase or snake_case `(i.e., subjectAge or subject_age)`
  - avoid using reserved words or built-in functions like `mean, TRUE, NA, FALSE etc.`
- code for human readibility (logical spaces and lines)

4 main types in R

1. **Character** or **string** has quotes around them `(i.e., "cat")`
2. **Logical** is a boolean `(i.e,`TRUE` or `FALSE` or NA)`
3. **Double** is a number `(i.e., `3.1`) Note: R's default for numbers`
4. **Integer** ex. 100 `(i.e., 100L) <- Note: specify integer using L`

use `typeof()` function to determine type of an object

can also check type using `is_*()` tidyverse function `(*i.e., is_logical(), is_character(),etc)`

Vectors

created using the `c()` function, can be logical, numeric, character, mixed

- explicit coercion; user explicitly converts one type of vector to another
  - use `as.*()` function in R `(*i.e., as.numeric(), as.logical(), as.character() etc)`
- implicit coercion; automatic conversion of vector from one type to another by the R without the need for explicit user instructions
  - in a mixed vector, R will always coerce to match the most "complex" type
    - character > double > integer > logical
- recycling; if operation requires a longer vector than what was given, R will recycle it
  - i.e., `1:2 + 1:4` gives the output `2 4 4 6` (Note: a warning will appear if the recycled vector is not a complete multiple of the larger vector)

    `1 2 1 2`

    `1 2 3 4`
- naming; can name elements within a vector which is helpful for subsetting
  - i.e., `myVector <- c(a=10,b=20,c=30)`

    `myVector[['a']]` will return 10 (can also subset with indexing `i.e., myVector[[1]])`

Lists

recursive vectors

- `str()` will return the structure of a list
- 3 ways to subset a list
  - i.e., `myList <- list(a = 7, b ="abc", c=FALSE)`
    1. `myList[1]` extracts a sublist. output would be

```
$a
[1] 7
```

2. `myList[[1]]` extracts a single component from a list, output would be

```
[1] 7
```

3. `my_List$a` is a shorthand for extracting a named element in a list; works similar to `[[]]`

```
[1] 7
```

Tibbles

in tidyverse we use tibbles instead of data frames; you can coerce a data set into a tibble by using the function `as_tibble()`

can subset a data frame using the %>% pipe operator

```
data_set %>%

.$column_name
```

pipe operator becomes useful when chaining together multiple functions (i.e., filtering, arranging, selecting columns in a data frame)

Strings

`stringr` library has multiple string functions we can use on strings! some include:

- `str_to_lower or str_to_UPPER` which converts all letters in a string to lower or upper case, respectively
- `str_c()` to combine strings; Note: can use the `sep =` argument to specify how you want to separate the strings

pattern matching using regular expressions

- `str_view()` takes a string vector and a regular expression and shows you where they match
  - i.e., `myFruit <- c("apple","pear","orange")`
    - `str_view(myFruit, "ap")` indicates where in the string “ap” is, gives output:

```
[1] | <ap>ple
```

    - `str_view(myFruit, ".a.")` indicates where in the string **any character (.)**, a then any character again. gives output:

```
[2] | p<ear>
[3] | o<ran>ge
```

  - *Note:* to search for a

- dot <- “\\.”
- slash <- “\\\\”

- **Anchors:**
  - ^ to match the start of a string
  - `str_view(myFruit, "^a")` indicates string where “a” is the first letter, gives output:

    ```
    [1] | <a>pple
    ```

  - $ to match the end of a string
  - `str_view(myFruit, "e$")` indicates string where “e” is the last letter, gives output:

    ```
    [1] | appl<e>
    [3] | orang<e>
    ```

- **Special patterns**
  - `.` matches any character

    ```
    > str_view(myFruit, ".")
    [1] | <a><p><p><l><e>
    [2] | <p><e><a><r>
    [3] | <o><r><a><n><g><e>
    ```

  - `\d` matches any digit

    ```
    > myDigits <- c(1,2,3)
    > str_view(myDigits, "\\d")
    [1] | <1>
    [2] | <2>
    [3] | <3>
    ```

  - `\s` matches any white space

    ```
    > myString <- c("Hello  ")
    > str_view(myString, "\\s")
    [1] | Hello< >< >
    ```

  - `[xyz]` matches x, y, or z

    ```
    > str_view(myFruit, "[aeo]")
    [1] | <a>ppl<e>
    [2] | p<e><a>r
    [3] | <o>r<a>ng<e>
    ```

  - `[^xyz]` matches anything except x, y, or z

```
> str_view(myFruit, "[^aeo]")
[1] │ a<p><p><l>e
[2] │ <p>ea<r>
[3] │ o<r>a<n><g>e
```

- **Repetitions: how many times a pattern matches**
  - `?` 0 or 1 time
  - `+` 1 or more
  - `*` 0 or more
  - `{n}` exactly how n times

    ```
    > str_view(myFruit, "p{2}") #pattern where p is repeated 2 times
    [1] │ a<pp>le
    ```

  - `{n,}` n or more times
  - `{,m}` at most m times
  - `{n,m}` between n and m times

    ```
    > str_view(myFruit, "p{1,2}") #pattern where p is repeated 1 to 2 times
    [1] │ a<pp>le
    [2] │ <p>ear
    ```

- By using parentheses, you can combine complex expressions i.e., `(..)\\1` matches any two consecutive repeated characters.
  - The `()` define a capturing group, which allows us to refer to the matched sequence later using backreferences.
  - `".."` represents any two characters.
  - `"\1"` backreference refers to the content of the first capturing group

- `str_detect()` determines if a string vector and a regular expression match, returning a logical vector

  ```
  > str_detect(myFruit, "p{1,2}")
  [1]  TRUE  TRUE FALSE
  ```

- `str_count()` determines how many matches there are in a string (1 = TRUE, 0 = FALSE)

  ```
  > str_count(myFruit, "p{1,2}")
  [1] 1 1 0
  ```

- `str_subset()` selects the elements that match a pattern

  ```
  > str_subset(myFruit, "p{1,2}")
  [1] "apple" "pear"
  ```