WebScraping 2

HS

Introduction

The Art of Data Rectangling

Data rectangling is about transforming data that's organized in a complex, layered (**hierarchical**) structure into a simple, table-like format with rows and columns. Think of it as turning a **tree-like structure** into a spreadsheet. This process is useful because many datasets, especially those from the web, are in this complex, nested format.

Key Concepts You'll Learn

1. **Lists**: In programming, a "list" is a way to organize data hierarchically, where each item can contain other lists or data. Lists make it possible to store complex data in layers, like folders within folders on a computer.
2. **Key Functions for Rectangling**:
   - `tidyr::unnest_longer()`: Expands list data by "stretching it out" vertically, adding more rows.
   - `tidyr::unnest_wider()`: Expands list data by "spreading it out" horizontally, adding more columns.
3. **JSON**: This is a common format for sharing data on the web. JSON data often comes in a nested format, so understanding lists and using functions like `unnest_longer` and `unnest_wider` helps us convert JSON data into easy-to-use tables.

With these tools, you'll be able to transform complex web data into a manageable format for analysis!

This is important because hierarchical data is surprisingly common, especially when working with data that comes from the web.

Install and Load Packages

Install and load `tidyverse`

- `tidyverse`: This is a collection of R packages (including tidyr) designed for data manipulation, cleaning, and visualization. tidyr, in particular, helps us organize messy data into a clean "tidy" format, making it easier to analyze.

```
library(tidyverse)
```

Install and load `repurrrsive`

- `repurrrsive`: This package provides example datasets for practicing "rectangling." Rectangling means transforming data from nested (like lists within lists) into a flat, rectangular shape (like a table). This package will give us interesting data to practice this skill.

```
library(repurrrsive)
```

Install and load `jsonlite`

- `jsonlitee`: This package helps us read and write JSON files. JSON is a common format for structured data, and `jsonlite` can turn JSON data into R lists so we can work with it in R.

```
library(jsonlite)
```

This setup will help you understand the basics of extracting, processing, and cleaning web data in R using these three packages. As you become more familiar with these tools, you'll be able to handle more complex web scraping tasks!

List

List

- When you're scraping data from the web, sometimes the information you collect isn't in a simple table. It might have lots of pieces grouped together, like a name, date, and description for each item. This is where lists in R become really handy.
- A *list* can hold different types of data all together—numbers, text, even other lists! Think of it as a "container" that lets you keep related data in one place, like organizing all the details of a product in an online store.

Naming Parts of a List

- It's often convenient to name the parts of a list to keep things clear. You can name them just like naming columns in a data frame or tibble. Here's a quick example:

```
product <- list(name = "Laptop",
                price = 1200, in_stock = TRUE)
product
```

```
$name
[1] "Laptop"

$price
[1] 1200

$in_stock
[1] TRUE
```

Here, we have a list `product` with named elements/components (also called "children")—`name`, `price`, and `in_stock`.

Checking the Structure with `str()`

The `str()` function gives you a quick, compact view of the list's structure. This is really useful because it shows you each child of the list on a new line with its name, type, and a sneak peek at its contents.

Example:

```
str(product)
```

```
List of 3
 $ name    : chr "Laptop"
 $ price   : num 1200
 $ in_stock: logi TRUE
```

Why This Matters in Web Scraping

- When scraping data, you often get a lot of mixed information. Organizing it in lists and then using `str()` to inspect the structure helps make sense of the data quickly.

Hierarchy

- **Lists** can contain any type of object, including other **lists**. This makes them suitable for representing `hierarchical` (`tree-like`) structures:

```
x3 <- list(list(1, 2), list(3, 4))
str(x3)
```

```
List of 2
 $ :List of 2
  ..$ : num 1
  ..$ : num 2
 $ :List of 2
  ..$ : num 3
  ..$ : num 4
```

- When **lists** become more complex (like a **list** inside a **list**), `str()` is helpful. It shows the structure of the **list**, letting you see its hierarchy at a glance. But if the **list** is really big and complex, you might want to use `View()` in RStudio. This gives an interactive view where you can click to expand different parts of the **list**. RStudio even helps by showing the code to access specific parts as you click.

```r
x5 <- list(1, list(2, list(3, list(4, list(5)))))
str(x5)
```

```
List of 2
 $ : num 1
 $ :List of 2
  ..$ : num 2
  ..$ :List of 2
  .. ..$ : num 3
  .. ..$ :List of 2
  .. .. ..$ : num 4
  .. .. ..$ :List of 1
  .. .. .. ..$ : num 5
```

- As **lists** get even larger and more complex, `str()` eventually starts to fail, and you'll need to switch to `View()`

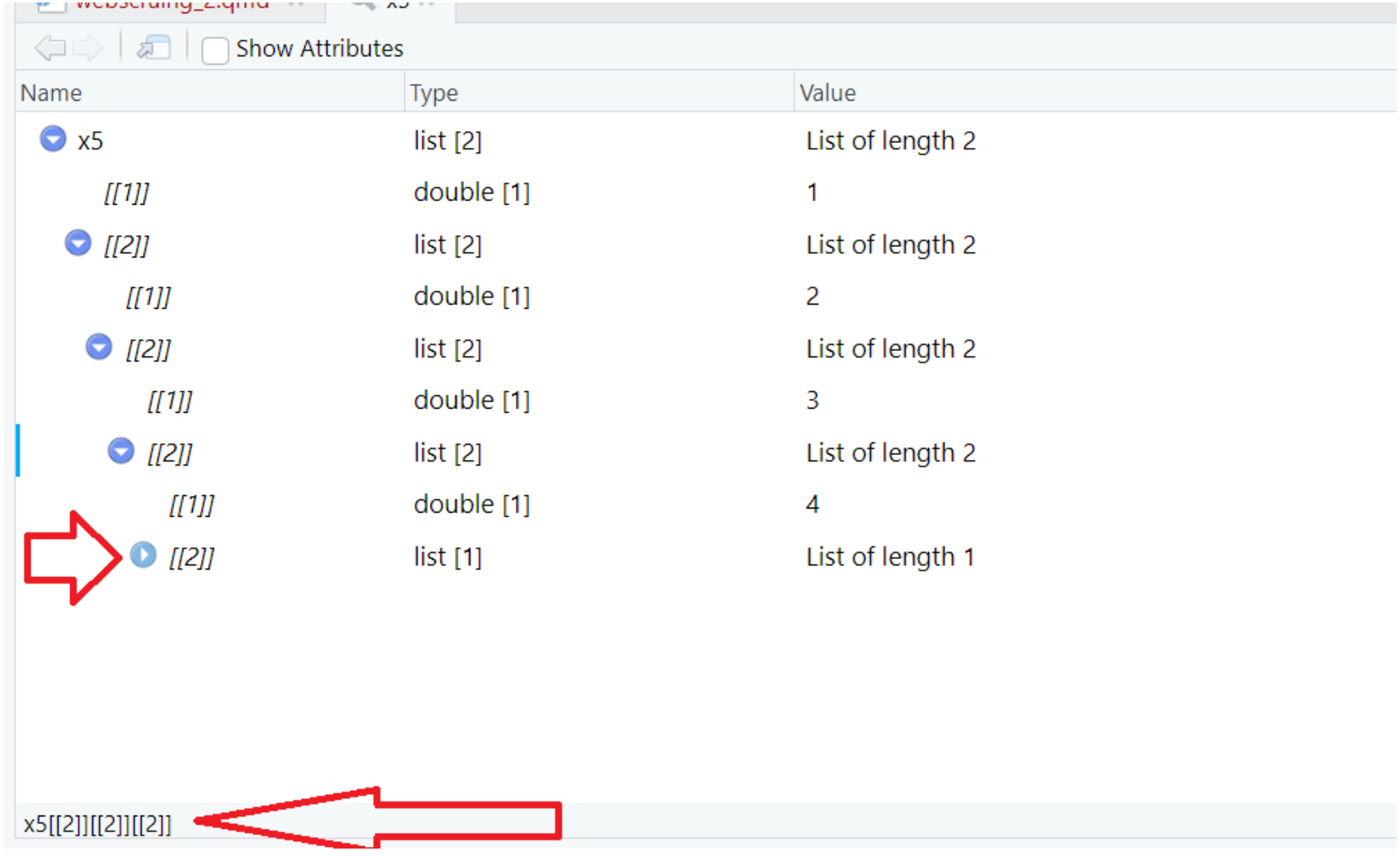
View()

```
View(x5)
```

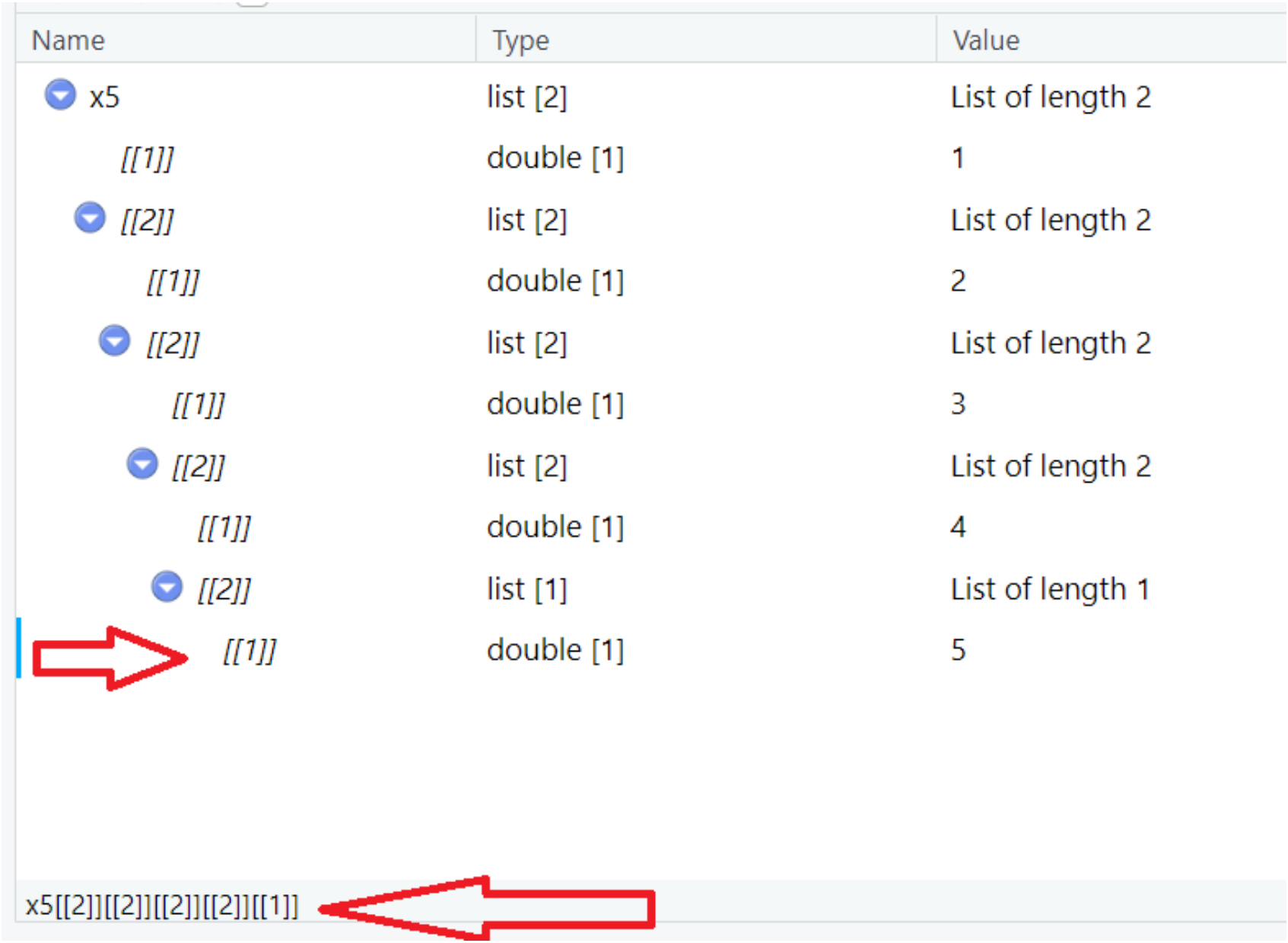
- The viewer starts by showing just the top level of the list, but you can interactively expand any of the components by clicking on the blue button on the left to expand the components.

Note the bottom-left corner: if you click an element of the list, RStudio will give you the subsetting code needed to access it, in this case

- Clicking on the `[[2]]` of the last blue button will provide `x5[[2]][[2]][[2]]` in the button line of RStudio



- Clicking on the `[[1]]` of the last blue button will provide `x5[[2]][[2]][[2]][[1]]` in the button line of RStudio.



List-columns

- In data frames (like a tibble), lists can be used as columns, called list-columns. .
  - Lists can also live inside a `tibble`, where we call them list-columns.
- List-columns are useful because they allow you to place objects in a tibble that wouldn't usually belong in there. In particular, list-columns are used a lot in the tidymodels ecosystem, because they allow you to store things like model outputs or resamples in a data frame.

Example

```
# Create a tibble with list-columns
df <- tibble(
  id = 1:3,
  data = list(
    c(1, 2, 3),    # First row contains a vector of 3 numbe
    c(4, 5),       # Second row contains a vector of 2 numbe
    c(6, 7, 8, 9) # Third row contains a vector of 4 numbe
  )
)

# Print the tibble
print(df)
```

- The values of column data are `list`

Another Example

```r
df <- tibble(
  x = 1:2,
  y = c("a", "b"),
  z = list(list(1, 2), list(3, 4, 5))
)
df
```

```
# A tibble: 2 x 3
      x y     z
  <int> <chr> <list>
1     1 a     <list [2]>
2     2 b     <list [3]>
```

- Computing with list-columns is harder, but that's because computing with lists is harder in general

```
df |>
  str()
```

```
tibble [2 x 3] (S3: tbl_df/tbl/data.frame)
 $ x: int [1:2] 1 2
 $ y: chr [1:2] "a" "b"
 $ z:List of 2
  ..$ :List of 2
  .. ..$ : num 1
  .. ..$ : num 2
  ..$ :List of 3
  .. ..$ : num 3
  .. ..$ : num 4
  .. ..$ : num 5
```

- Pulling only z component

```
df$z
```

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

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



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

[[2]][[2]]
[1] 4

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

```r
df$z %>%
  str()
```

```
List of 2
 $ :List of 2
  ..$ : num 1
  ..$ : num 2
 $ :List of 3
  ..$ : num 3
  ..$ : num 4
  ..$ : num 5
```

- Pulling only z component, using str()for compact output

```
df |>
  pull(z) |>
  str()
```

```
List of 2
 $ :List of 2
  ..$ : num 1
  ..$ : num 2
 $ :List of 3
  ..$ : num 3
  ..$ : num 4
  ..$ : num 5
```

- *You may use {base} package to create list-column. But it's easier to use list-columns with tibbles because `tibble()` treats lists like vectors.*

Unnesting

- In web scraping with R, “unnesting” is a way to simplify complex data structures, especially lists within data frames. When you unnest a list-column, you expand it into multiple columns or rows, making the data easier to work with.
- **There are two main types of list-columns:**
  - Named List-Columns
  - Unnamed List-Columns

Named List-Columns

These contain lists where each element has a name (like “name,” “age,” etc.), and these names are the same across all rows. When you unnest named list-columns, each named element becomes its own column, so you end up with a new data frame with separate columns for each item in the list.

```
df1 <- tribble(
  ~x, ~y,
  1, list(a = 11, b = 12, c= 18),
  2, list(a = 21, b = 22, c= 28),
  3, list(a = 31, b = 32, c= 38)
)
df1
```

Unnamed List-Columns

These are simpler and don't have named elements. When you unnest these, they usually expand into rows, not columns.We'll get one row for each child.

```
df2 <- tribble(
  ~x, ~y,
  1, list(11, 12, 13),
  2, list(21),
  3, list(31, 32),
)
df2
```

tidyr Package

- {tidyr} provides two functions for these two cases:
  - unnest_wider() and
  - unnest_longer().

unnest_wider()

- This function is used when you have a list-column where each item in the list should be spread across multiple new columns. It's like taking a list of things and breaking them into individual columns, making the data wider.

```
df1 |>
  unnest_wider(y)  # y is the column contains lists
```

```
# A tibble: 3 x 4
      x     a     b     c
  <dbl> <dbl> <dbl> <dbl>
1     1    11    12    18
2     2    21    22    28
3     3    31    32    38
```

- By default, the names of the new columns come exclusively from the names of the list elements, but you can use the names_sep argument to request that they combine the column name and the element name. This is useful for disambiguating repeated names.

```
df1 |>
  unnest_wider(y)
```

```
# A tibble: 3 x 4
      x     a     b     c
  <dbl> <dbl> <dbl> <dbl>
1     1    11    12    18
2     2    21    22    28
3     3    31    32    38
```

```r
df1 |>
  unnest_wider(y, names_sep = "_")
```

```
# A tibble: 3 x 4
      x   y_a   y_b   y_c
  <dbl> <dbl> <dbl> <dbl>
1     1    11    12    18
2     2    21    22    28
3     3    31    32    38
```

unnest_longer()

- This function is for when you have a list-column and you want to turn each item in the list into a new row. It's like taking a list of things and stacking them vertically, making the data longer.

```
df2 |>
  unnest_longer(y)
```

```r
df2 |>
  unnest_longer(y)
```

```
# A tibble: 6 x 2
      x     y
  <dbl> <dbl>
1     1    11
2     1    12
3     1    13
4     2    21
5     3    31
6     3    32
```

- **What happens if one of the elements is empty?**

```
df6 <- tribble(
  ~x, ~y,
  "a", list(1, 2),
  "b", list(3),
  "c", list()
)
df6 |> unnest_longer(y)
```

```
# A tibble: 3 x 2
  x         y
  <chr> <dbl>
1 a         1
2 a         2
3 b         3
```

- We get zero rows in the output, so the row effectively disappears.

- If you want to preserve that row, adding `NA` in y, `setkeep_empty = TRUE`.

```
df6 |> unnest_longer(y, keep_empty = TRUE)
```

```
# A tibble: 4 x 2
  x           y
  <chr> <dbl>
1 a           1
2 a           2
3 b           3
4 c          NA
```

Inconsistent types

- **What happens if you unnest a list-column that contains different types of vector?**

```
df4 <- tribble(
  ~x, ~y,
  "a", list(1),
  "b", list("a", TRUE, 5)
)
df4
```

```
# A tibble: 2 x 2
  x     y
  <chr> <list>
1 a     <list [1]>
2 b     <list [3]>
```

- `unnest_longer()` always keeps the set of columns unchanged, while changing the number of rows. So what happens? How does unnest_longer()produce five rows while keeping everything in y?

```
df4 |>
  unnest_longer(y)
```

```
# A tibble: 4 x 2
  x     y
  <chr> <list>
1 a     <dbl [1]>
2 b     <chr [1]>
3 b     <lgl [1]>
4 b     <dbl [1]>
```

- Because `unnest_longer()` can't find a common type of vector, it keeps the original types in a list-column.
  - You might wonder if this breaks the commandment that every element of a column must be the same type. It doesn't: every element is a list, even though the contents are of different types.

Other functions

- `unnest_auto()` automatically picks between `unnest_longer()` and `unnest_wider()` based on the structure of the list-column. It's great for rapid exploration, but ultimately it's a bad idea because it doesn't force you to understand how your data is structured, and makes your code harder to understand.

```
df1 |>
  unnest_auto(y)
```

```
# A tibble: 3 x 4
      x     a     b     c
  <dbl> <dbl> <dbl> <dbl>
1     1    11    12    18
2     2    21    22    28
3     3    31    32    38
```

```r
df2 |>
  unnest_auto(y)
```

```
# A tibble: 6 x 2
      x     y
  <dbl> <dbl>
1     1    11
2     1    12
3     1    13
4     2    21
5     3    31
6     3    32
```

- `unnest()` expands both rows and columns. It's useful when you have a list-column that contains a 2d structure like a data frame, which you don't see here, but you might encounter if you use the {tidymodels} ecosystem.

```r
# Sample data with nested lists
data <- tibble(
  id = 1:3,
  info = list(
    tibble(score = c(10, 20), time = c(5, 10)),
    tibble(score = c(15, 25, 35), time = c(7, 8, 9)),
    tibble(score = c(40), time = c(12))
  )
)

data
```

```
# A tibble: 3 x 2
     id info
  <int> <list>
1     1 <tibble [2 x 2]>
2     2 <tibble [3 x 2]>
3     3 <tibble [1 x 2]>
```

```r
# Using unnest() to expand the 'info' column
data %>%
  unnest(info)
```

```
# A tibble: 6 x 3
     id score  time
  <int> <dbl> <dbl>
1     1    10     5
2     1    20    10
3     2    15     7
4     2    25     8
5     2    35     9
6     3    40    12
```

JSON

Scalars

- When web scraping, you often deal with data in JSON (JavaScript Object Notation) format. It's a simple and lightweight data format designed for machines to read and write easily. Here's a quick breakdown of how JSON works:
  - `Null`: Similar to NA in R, it means "no data" or "empty." For example, null is used when a value is missing.
  - `String`: A string in JSON is like a text in R, but it must be enclosed in **double quotes**. - Example: "hello world".
  - `Number`: JSON numbers are like R numbers. They can be integers (123), decimals (123.45), or in scientific notation (1.23e3). However, JSON **doesn't support** special values like `Inf` or `NaN`.
  - `Boolean`: Similar to R's TRUE and FALSE, but in JSON, they are written as `true` or `false`.
- **The symbols "NaN", "Infinity", and "-Infinity" are replaced with "null"**

Complex Types

- `JSON` can represent more than just simple values. To handle multiple values, it uses two structures:
  - `Arrays`
  - `Objects`

Arrays

- These are like unnamed lists in R. An array is a collection of values, enclosed in square brackets [].
  - **Example:**

1. [“apple”, “banana”, “cherry”].
2. [null, 1, “string”, false]

Objects

- These are like named lists in R. An object contains pairs of "keys" and "values" enclosed in curly braces {}.
  - **Example:** {"name": "John", "age": 30}.

Working with Dates and Times in JSON

- JSON doesn't have a special way to store dates or times. Instead, dates are usually *stored as strings (text)*. For example:

```
d <- "2024-11-06"
class(d)
```

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

- If you want to use these as dates in R, you need to convert them using functions like `readr::parse_date()` for dates or `readr::parse_datetime()` for dates with times.
  - Converts the string into a date

```
library(readr)
date <- parse_date("2024-11-06")
 class(date)
```

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

- Converts the string into a date-time

```
datetime <- parse_datetime("2024-11-06 14:30:00")

class(datetime)
```

```
[1] "POSIXct" "POSIXt"
```

Working with Numbers in JSON

- Sometimes, numbers in JSON are not stored as proper numbers, but as strings (text). This can happen because JSON's rules for numbers aren't always perfect.

For example, a number might look like this:

```
n <- "123.45"  # Stored as a string
class(n)
```

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

- To turn it into a proper number in R, you can use `readr::parse_double()`.

```
number <- parse_double("123.45")

class(number)
```

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

- **So, when working with JSON data, make sure to use the right functions to convert dates and numbers into proper R formats!**

{jsonlite}

- When working with JSON data in R, there are two main functions you need to know about:
  - `read_json()`: Use read_json() to read a JSON file from disk.
  - `parse_json()`

A path to a json file inside the package

- The {repurrrsive} package also provides the source for *gh_user as a JSON file*.

```
library(repurrrsive)
gh_users_json()   #from {repurrsive}
```

```
[1] "C:/Users/semiyari/AppData/Local/R/win-library/4.4/repu
```

`read_json()`

- This function is used to read a JSON file from your computer (disk). It loads the data from a JSON file so you can work with it in R.

Example: The {`repurrsive`} package also provides the source for *`gh_user` as a JSON file* and you can read it with `read_json()`:

```
gh_users2 <- read_json(gh_users_json())
```

View The File

- View the file

```
gh_users2 |>
  View()
```

```
Error in (function (..., row.names = NULL, check.rows = FAI
```

```
gh_users |>
  View()
```

```
Error in (function (..., row.names = NULL, check.rows = FAI
```

Check if they are the same

```
identical(gh_users, gh_users2)
```

```
[1] TRUE
```

`parse_json()`

- The function `parse_json()` is used to convert a string of JSON data into an R object (something that R can work with).

Example 1: JSON with a Simple Number

- Let's start with the simplest JSON: a single number (in JSON, it looks like 1).

```
str(parse_json('1'))
```

```
int 1
```

- The string '1' is JSON that represents the number 1.
- When you parse it using `parse_json()`, R understands it as an integer (int in R).

Example 2: JSON with an Array

- Now, let's deal with a list of numbers (in JSON, it looks like [1, 2, 3]).

```
str(parse_json('[1, 2, 3]'))
```

```
List of 3
 $ : int 1
 $ : int 2
 $ : int 3
```

- The string '[1, 2, 3]' is a JSON array containing three numbers.
- When you parse it, R turns this array into a list of three integers.

Example 3: JSON with an Object

- Let's try JSON that represents an object (like a dictionary). In this case, it has a key x that holds an array of numbers.

```
str(parse_json('{"x": [1, 2, 3]}'))
```

```
List of 1
 $ x:List of 3
  ..$ : int 1
  ..$ : int 2
  ..$ : int 3
```

- The string '{"x": [1, 2, 3]}' is a JSON object where the key x has an array [1, 2, 3] as its value.
- When parsed, R understands it as a list with one element, where the element x is a list of numbers.
- This means `parse_json('{"x": [1, 2, 3]}')` converted the JSON object into a list with one key (`x`), and the value of `x`is a list of three numbers: 1, 2, and 3.

- In all of these cases, `parse_json()` is turning a JSON string into something that R can understand and use in your code (like numbers or lists).

Starting the rectangling process

- In most cases, JSON files contain one big top-level array at the beginning. This array holds data about many items, like multiple pages, records, or results. Think of it like a list of items, where each item has its own details.
- Here's an example of what that JSON might look like:

```
[
    {"name": "John", "age": 30},
    {"name": "Jane", "age": 25},
    {"name": "Sam", "age": 35}
]
```

- Here, we have a top-level array `[ ]` that contains multiple records (each person's data). Each record inside the array has its own `name` and `age`.

Converting JSON to a Table in R

- In R, we can use the `tibble()` function to convert this JSON array into a table format. Each item in the array will become a row in the table:

Example

```
json <- '[
  {"name": "John", "age": 34},
  {"name": "Jane", "age": 25},
  {"name": "Sam", "age": 27}
]'
df <- tibble(json = parse_json(json))
df
```

```
# A tibble: 3 x 1
  json
  <list>
1 <named list [2]>
2 <named list [2]>
3 <named list [2]>
```

- This will give us a table where:
  - Each row represents a person (John, Jane, Sam).
  - Each column represents their attributes (name and age).

```r
df |>
  unnest_wider(json)
```

```
# A tibble: 3 x 2
  name    age
  <chr> <int>
1 John     34
2 Jane     25
3 Sam      27
```

How about if we take unnest_auto()?

```
df |>
  unnest_auto(json)
```

```
# A tibble: 3 x 2
  name    age
  <chr> <int>
1 John     34
2 Jane     25
3 Sam      27
```

- `unnest_wider()` takes each element in the json column (which is a list) and splits it into multiple columns.
- In this example, each JSON object (e.g., {“name”: “John”, “age”: 34}) contains two fields: `name` and age.
- `unnest_wider(json)` will create separate columns for `name` and age.

Rare Cases

- Sometimes, a JSON file is structured as a single "top-level" object, which represents just one main item or "thing." In R, to work with this kind of JSON data in a nice, table-like format (like a tibble), you need to make a small adjustment first.
- Here's how it works:

Step 1: Wrap JSON in a List

- **{} are wrapped by ' (single quotation)**

```
 '{
  "status": "OK",
  "results": [
    {"name": "John", "age": 34},
    {"name": "Susan", "age": 27}
  ]
}'
```

```
[1] "{\n  \"status\": \"OK\", \n  \"results\": [\n    {\"na
```

- This JSON data has a single top-level object with two keys: "status" and "results".
  - "results" is a list of two items, each with `name` and age fields.

Step 2: Create a Tibble

```
json <- '{
  "status": "OK",
  "results": [
    {"name": "John", "age": 34},
    {"name": "Susan", "age": 27}
 ]
}
'

df <- tibble(json = list(parse_json(json)))
df
```

```
# A tibble: 1 x 1
  json
  <list>
1 <named list [2]>
```

Step 3: Unnest

```
df <- tibble(results = parse_json(json)$results)
df |>
  unnest_wider(results)
```

```
# A tibble: 2 x 2
  name    age
  <chr> <int>
1 John     34
2 Susan    27
```

What This Code Does:

- `fromJSON(json)` reads the JSON data.
- `tibble(json = list(parse_json(json)))`puts the data into a tibble.
- `toJSON()` Convert dataframe to JSON

`fromJSON()`

- Often works well, particularly in simple cases, but we think you're better off doing the rectangling yourself so you know exactly what's happening and can more easily handle the most complicated nested structures.

Examle: Most Cases

```
json <- '[
  {"name": "John", "age": 34},
  {"name": "Susan", "age": 27}
]'

fromJSON(txt = json)
```

```
   name age
1  John  34
2 Susan  27
```

EXample: Rare Cases

```
json <- '{
  "status": "OK",
  "results": [
    {"name": "John", "age": 34},
    {"name": "Susan", "age": 27}
 ]
}
'

fromJSON(json, simplifyVector = TRUE)
```

```
$status
[1] "OK"

$results
   name age
1  John  34
2 Susan  27
```

tibble(json = list(parse_json(json)))

```
json <- '{
  "status": "OK",
  "results": [
    {"name": "John", "age": 34},
    {"name": "Susan", "age": 27}
 ]
}
'

df <- tibble(json = list(parse_json(json)))
df
```

```
# A tibble: 1 x 1
  json
  <list>
1 <named list [2]>
```

toJSON()

- `toJSON()` Convert dataframe to JSON
- Now `df` is a data frame and we turn it back to JSON.

```
toJSON(df)
```

```
[{"json":{"status":["OK"],"results":[{"name":["John"],"age"
```

JSON is the most common data format returned by web APIs.

- What happens if the website doesn't have an API, but you can see data you want on the website?
- Please see the "2_scrape.html" and "3_scrape.html": web scraping, extracting data from HTML webpages.