Review of the course "R for Data Science" Part 02(Talk 05~ 08)

By Haoran Nie @ HUST Life ST

To reduce the size, all the codes listed will ***NOT*** include the output as picture.

Contents

R for bioinformatics, data wrangler, part 1

Talk 05

TOC

Pipe in R

What is pipe in R?

- Pipe is `%>%`.
- It comes from the `magrittr` package by **Stefan Milton Bache**.
- Packages in the `tidyverse` load `%\>%` for you automatically, so you don't usually load `magrittr` explicitly.
- The essence is the passing of intermediate values.

Example

```
library(tidyverse)
library(magrittr)
a =
 subset(swiss, Fertility > 20)
cor.test(a$Fertility, a$Education)
```

The code above can be replaced by:

```
swiss %>%
  subset(., Fertility > 20) %$%
  cor.test(Education, Fertility)
```

You should remember that almost **all** the funtions support pipe.

Other kinds of pipe

- `%T>%`: Return left-side values
- `%$>%`: Attach ...
- `%<>%`

ATTENTION

1. The use of pipe makes the idea clearer;
2. Therefore, try to use `%>%` (which has a clear direction) instead of the other pipe, which has an unclear direction.

Data Wrangler - `dplyr`

What is `dplyr`?

- The next iteration of `plyr`,
- Focusing on only data frames (also tibble),
- Row-based manipulation,
- `dplyr` is faster and has a more consistent API.

`dplyr` provides a consistent set of verbs that help you **solve the most common data manipulation challenges**:

- `select()` Select columns, based on column name rules.
- `filter()` Filter rows by rule.
- `mutate()` Add new column, calculated from other columns (no change in number of rows).
- `summarise()` Converts multiple values to a single value (via mean, median, sd, etc.), generating new columns (total number of rows is reduced, usually in conjunction with `group_by`).
- `arrange()` Sorting the rows.

```
# Read the file
library(tidyverse)
mouse.tibble =
  read_delim(
    file = "data/mouse_genes_biomart_sep2018.txt",
    delim = "\t",
    quote = "",
    show_col_types = FALSE
  )

# View mouse.tibble content
ttype.stats =
  mouse.tibble %>%
    count(`Transcript type`) %>%
    arrange(-n)

# View mouse.tibble content, cont.
chr.stats =
  mouse.tibble %>%
    count(`Chromosome/scaffold name`) %>%
    arrange(-n)
```

R for bioinformatics, data wrangler, part 2

Talk 06

TOC

`tidyr`

- `pivot_longer()` to take the place of `gather`
- `pivot_wider()` to take the place of `spread`

Data Wrangler - `tidyr`

You can get `tidyr` in the package set `tidyverse`, or simply install it the first time you want to use it via `install.packages("tidyr")`.

The usage of `tidyr`

- Interconversion of wide and long data

```
# Eg 1
library(tidyverse)
grades2 =
        read_tsv(file = "data/grades2.txt")

grades3 =
        grades2 %>%
        pivot_longer(
    - name,
    names_to = "course",
    values_to = "grade"
  )

# Eg 2
grades3_wide = grades3_long %>%
  pivot_wider(
    names_from = "course",
    values_from = "grade"
  )
```

What's the difference between wide and long data?

Here are pros and cons of wide data:

- Pros:
- Natural and easy to understand;
- Cons
- Not easy to handle;
    - More problematic when sparse.

If you meet `NA` in the 1st example, you can do like this:

```
grades3_1 =
        grades3[!is.na(grades3$grade), ]
grades3_2 =
        grades3[complete.cases(grades3), ]

# A better solution
grades3_long = grades2 %>%
  pivot_longer( - name,
                names_to = "course",
                values_to = "grade",
                values_drop_na = TRUE)

# Pay attention to the variant named "values_drop_na"
```

More functions in `tidyr`: (See @ https://r4ds.hadley.nz/data-tidy.html)

- `tidyr::separate()`

Usage:

```
separate(
  data,
  col,
  into,
  sep = "[^[:alnum:]]+",
  remove = TRUE,
  convert = FALSE,
  extra = "warn",
  fill = "warn",
  ...
)

# Default parameters are listed.
```

- `tidyr::unite()`

Usage:

```
unite(
  data,
  col,
  ...,
  sep = "_",
  remove = TRUE,
  na.rm = FALSE
)

# Default parameters are listed.
```

R for bioinformatics, Strings and regular expression

Talk 07

TOC

`stringr`

1. basics

- length
- uppercase, lowercase
- unite, separate
- string comparisons, sub string

1. regular expression

Note that each link listed below provides many further links to other interesting facilities and topics, you can ***ONLY*** open the links through VSCode built-in WebView. Sorry for the temporary inconvenience, but due to time constraints I can only fix this bug *in the next release*, please contact me if you need help.

Basics

Before you start...

```
library(stringr)
```

Also notice other famous packages used to manipulating string:

`stringi`(Following are based on the official R Documentation)

Description `stringi` is THE R package for fast, correct, consistent, and convenient string/text manipulation. It gives predictable results on every platform, in each locale, and under any native character encoding.

Facilities available Refer to the following:

- about_search for string searching facilities; these include pattern searching, matching, string splitting, and so on. The following independent search engines are provided:
- about_search_regex – with ICU (Java-like) regular expressions,
  - about_search_fixed – fast, locale-independent, byte-wise pattern matching,
  - about_search_coll – locale-aware pattern matching for natural language processing tasks,
  - about_search_charclass – seeking elements of particular character classes, like "all whitespaces" or "all digits",
  - about_search_boundaries – text boundary analysis.

- `stri_datetime_format` for date/time formatting and parsing. Also refer to the links therein for other date/time/time zone- related operations.
- `stri_stats_general` and `stri_stats_latex` for gathering some fancy statistics on a character vector's contents.
- `stri_join`, `stri_dup`, `%s+%`, and `stri_flatten` for concatenation-based operations.
- `stri_sub` for extracting and replacing substrings, and `stri_reverse` for a joyful function to reverse all code points in a string.
- `stri_length` (among others) for determining the number of code points in a string. See also `stri_count_boundaries` for counting the number of Unicode characters and `stri_width` for approximating the width of a string.
- `stri_trim` (among others) for trimming characters from the beginning or/and end of a string, see also about_search_charclass, and `stri_pad` for padding strings so that they are of the same width. Additionally, `stri_wrap` wraps text into lines.
- `stri_trans_tolower` (among others) for case mapping, i.e., conversion to lower, UPPER, or Title Case, `stri_trans_nfc` (among others) for Unicode normalization, `stri_trans_char` for translating individual code points, and `stri_trans_general` for other universal text transforms, including transliteration.
- `stri_cmp`, `%s<%`, `stri_order`, `stri_sort`, `stri_rank`, `stri_unique`, and `stri_duplicated` for collation-based, locale-aware operations, see also about_locale.
- `stri_split_lines` (among others) to split a string into text lines.
- `stri_escape_unicode` (among others) for escaping some code points.
- `stri_rand_strings`, `stri_rand_shuffle`, and `stri_rand_lipsum` for generating (pseudo)random strings.
- `stri_read_raw`, `stri_read_lines`, and `stri_write_lines` for reading and writing text files.

Usage of `writeLines()` (from official R Documentation)

Description Write text lines to a connection.

Usage

```
writeLines(text, con = stdout(), sep = "\n", useBytes = FALSE)
```

Arguments

`text`	A character vector
`con`	A connection object or a character string.
`sep`	character string. A string to be written to the connection after each line of text.
`useBytes`	logical. See 'Details'.

Details If the `con` is a character string, the function calls `file` to obtain a file connection which is opened for the duration of the function call. (tilde expansion of the file path is done by `file`.)

If the connection is open it is written from its current position. If it is not open, it is opened for the duration of the call in `"wt"` mode and then closed again.

Normally `writeLines` is used with a text-mode connection, and the default separator is converted to the normal separator for that platform (LF on Unix/Linux, CRLF on Windows). For more control, open a binary connection and specify the precise value you want written to the file in `sep`. For even more control, use `writeChar` on a binary connection.

`useBytes` is for expert use. Normally (when false) character strings with marked encodings are converted to the current encoding before being passed to the connection (which might do further re-encoding). `useBytes = TRUE` suppresses the re-encoding of marked strings so they are passed byte-by-byte to the connection: this can be useful when strings have already been re-encoded by e.g. `iconv`. (It is invoked automatically for strings with marked encoding `"bytes"`.)

Difference between double quote("") and single quote(‘’)

In R and its string manipulation package `stringr`, there is no difference between strings defined with double quotes (") and single quotes ('). Both are used to define strings and you can use either depending on your preference or the situation.

For instance, if your string contains a single quote, you might find it easier to enclose the string in double quotes, and vice versa. Here's an example:

```
# Using double quotes when the string contains a single quote
string1 = "It's a beautiful day"

# Using single quotes when the string contains a double quote
string2 = 'He said, "Hello, world!"'
```

In both cases, R will interpret the contents between the quotes as a string.

Some of the functions in the stringr package are similar in function to those that come with the system.

Here are examples comparing some functions from the `stringr` package with their counterparts from base R:

1. **`str_length()` vs. `nchar()`:**

```
library(stringr)

# Using str_length from stringr
string = c("apple", NA, "banana", "")
str_length(string)
# Output: 5   NA   6   0

# Using nchar from base R
nchar(string)
# Output: 5  NA   6   0
```

Both `str_length()` and `nchar()` count the number of characters in each string element. However, `str_length()` handles missing values (`NA`) more consistently by returning `NA`, whereas `nchar()` might treat `NA` differently in certain cases.

1. **`str_sub()` vs. `substr()`:**

```
# Using str_sub from stringr
string = c("hello", "world", "example")
str_sub(string, start = 2, end = 4)
# Output: "ell" "orl" "xam"

# Using substr from base R
substr(string, start = 2, stop = 4)
# Output: "ell" "orl" "xam"
```

Both `str_sub()` and `substr()` extract substrings based on specified start and end positions. However, `str_sub()` allows negative indices to count from the end of the string, and it handles missing values more consistently.

1. **`str_replace()` vs. `sub()` or `gsub()`:**

```
# Using str_replace from stringr
string = c("apple pie", "banana bread", "cherry cake")
str_replace(string, pattern = "a", replacement = "X")
# Output: "Xpple pie" "bXnXnX breXd" "chXrry cXke"

# Using sub from base R
sub(pattern = "a", replacement = "X", x = string)
# Output: "Xpple pie" "bXnana bread" "cherry cake"
```

Both `str_replace()` and `sub()` are used for replacing parts of a string. However, `str_replace()` has a more intuitive interface and handles missing values more gracefully compared to `sub()`.

These examples demonstrate how `stringr` functions can be more consistent and user-friendly in handling various string operations compared to their base R counterparts.

Some of the functions in the `stringi` package are similar in function to those that come with the system.

Here are some functions in the `stringi` package that share similar functionalities with base R's string functions, along with examples showcasing their differences:

1. **`stri_length()` vs. `nchar()`**:
    - `stri_length()` in `stringi` calculates the number of code points in a string, accounting for Unicode characters.
    - `nchar()` in base R counts the number of characters in a string, but it might not handle Unicode characters as accurately as `stri_length()`.

```
library(stringi)

# Using stri_length from stringi
string = "café"
stri_length(string)
# Output: 4

# Using nchar from base R
nchar(string)
# Output: 4
```

In this example, both `stri_length()` and `nchar()` return the same count for ASCII characters. However, when dealing with Unicode characters, `stri_length()` can accurately count them as individual code points, whereas `nchar()` might not handle them correctly.

2. **`stri_split_*()` vs. `strsplit()`**:

   - `stri_split_*()` functions in `stringi` split a string based on various criteria like fixed patterns, regular expressions, or character classes.
   - `strsplit()` in base R performs a similar operation but might differ in handling certain edge cases and Unicode characters.

```
# Using stri_split_* from stringi
string = "apple, orange, café"
stri_split_fixed(string, pattern = ", ")
# Output: list("apple", "orange", "café")

# Using strsplit from base R
strsplit(string, split = ", ")
# Output: list("apple", "orange", "caf", "é")
```

   Here, `stri_split_fixed()` correctly splits the string, including the accented character "é," while `strsplit()` treats the accented "é" as two separate characters due to how it handles Unicode.

3. **`stri_detect()` vs. `grepl()`**:

   - `stri_detect()` in `stringi` checks if a pattern exists in a string and returns a logical value.
   - `grepl()` in base R performs a similar task but might differ in its handling of Unicode characters and certain pattern matching options.

```
# Using stri_detect from stringi
string = "This is a café"
stri_detect(string, regex = "café")
# Output: TRUE

# Using grepl from base R
grepl("café", string)
# Output: FALSE
```

   In this example, `stri_detect()` correctly detects the presence of the word "café," while `grepl()` returns a different result due to potential differences in Unicode handling or pattern matching options.

The examples highlight how `stringi` functions like `stri_length()`, `stri_split_*()`, and `stri_detect()` differ from their base R counterparts (`nchar()`, `strsplit()`, and `grepl()`) by providing more accurate handling of Unicode characters and often more versatile string manipulation options.

(In the slide) Difference between `toupper()`, `tolower()` and `stri_reverse()`

The functions `toupper()` and `tolower()` in base R and `stri_reverse()` in the `stringi` package perform similar tasks, but there are some differences in their functionality and usage:

1. **`toupper()` and `tolower()` in Base R:**

   - `toupper()` converts characters in a string to uppercase.
   - `tolower()` converts characters in a string to lowercase.

```r
# Using toupper and tolower from base R
string = "Hello World!"

toupper(string)
# Output: "HELLO WORLD!"

tolower(string)
# Output: "hello world!"
```

These functions are straightforward and work well for ASCII characters, converting them to uppercase or lowercase, respectively. However, they might not handle Unicode characters or locale-specific transformations.

2. **`stri_reverse()` in `stringi`:**

   - `stri_reverse()` reverses the order of characters in a string, including handling multibyte characters and Unicode sequences.

```r
library(stringi)

# Using stri_reverse from stringi
string = "café"

stri_reverse(string)
# Output: "éfac"
```

`stri_reverse()` reverses the characters in the string accurately, even when dealing with Unicode characters or multibyte sequences. It ensures correct reversal of characters irrespective of their encoding.

The key distinction lies in the handling of character cases and character sequence reversal. While `toupper()` and `tolower()` focus on case transformations for ASCII characters, `stri_reverse()` in `stringi` concentrates on accurately reversing character sequences, making it more suitable for handling multibyte characters and Unicode strings.

Tricks

- `stringi` The functions in the package all start with `stri_`.
- `strinr` starts with `str_`.

Regex - Regular Expression

1. Character classes:What characters are (not) matched?

```r
# Example 01
"abc_123_??$$^" %>% str_extract("\\s+") # Does this string include spaces?
"abc_123_??$$^" %>% str_extract("\\d+") # Numbers?
"abc_123_??$$^" %>% str_extract("\\w+") # [A-z0-9_]
```

`str_extract` : Take out the first match.

1. Matching position

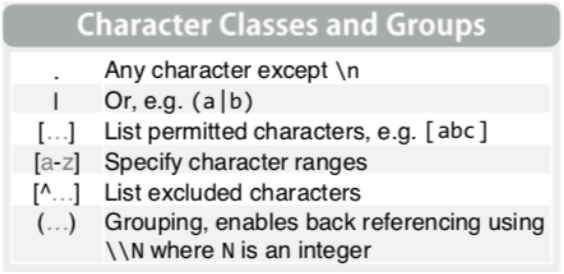
```r
# Example 02
# STRING ending in 'wei'
c("chen wei hua", "chen wei", "chen") %>% str_subset("wei$")

# CHARACTER ending in 'wei'
c("chen wei hua", "chen wei", "chen") %>% str_subset("wei\\b")
```

1. Number of matches

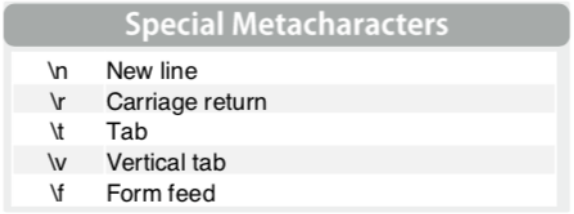
```r
# Example 03
"1234abc" %>% str_extract("\\d+")
"1234abc" %>% str_extract("\\d{3}")
"1234abc" %>% str_extract("\\d{5,6}")
"1234abc" %>% str_extract("\\d{2,6}")
```

1. Classes and groups



Character Classes and Groups	
.	Any character except \n
|	Or, e.g. (a|b)
[…]	List permitted characters, e.g. [abc]
[a-z]	Specify character ranges
[^…]	List excluded characters
(…)	Grouping, enables back referencing using \\N where N is an integer

2. Special characters



Special Metacharacters	
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed

R for bioinformatics, data iteration & parallel computing

Talk 08

TOC

- for loop
- `apply` functions
- The essence of `dplyr` is traversal.
- `map` functions in `purrr` package
- Iteration and Parallel Computing

Iteration Basics

`for loop` , getting data ready

Lookat this example:

```
df =
        tibble(
    a = rnorm(100),
    b = rnorm(100),
    c = rnorm(100),
    d = rnorm(100)
  )

# Calculate row means
res1 =
        vector("double", nrow(df))
for(row_idx in 1:nrow(df)){
  res1[row_idx] =
          mean( as.numeric(df[row_idx, ]))
}

res2 = c()
for(row_idx in 1:nrow(df)){
  res2[length(res2) + 1] =
          mean(as.numeric(df[row_idx, ]))
}

# Similar to Python

# Calculate column means
res2 =
        vector("double", ncol(df))
for(col_idx in 1:ncol(df)){
  res2[col_idx] =
          mean(df[[col_idx]])
}
```

You can replace it with `for loop`:

```
rowMeans(df)
colMeans(df)
```

Here are some other functions:

```
rowSums(df)
colSums(df)
```

`apply` functions

You can use `apply` with customizable function.

```
df %>% apply(
  .,
  2,
```

```
  function(x) {
    return(
      c(
        n = length(x),
        mean = mean(x),
        median = median(x)
      )
    )
  }
)
```

Something about `tapply()`:

The `tapply()` function in R is used to apply a function over subsets of a vector, splitting it by a factor or list of factors. It stands for "table apply" and is particularly useful for summarizing data by groups or categories.

Here's a breakdown of its usage:

```
tapply(X, INDEX, FUN)
```

- `X`: The vector (or array) on which you want to apply the function.
- `INDEX`: A factor or list of factors that define the groups. These factors determine how the vector `X` is split.
- `FUN`: The function to be applied to each subset of `X`.

For example:

```
# Creating a sample vector and a corresponding factor
values = c(10, 20, 30, 40, 50)
categories = factor(c("A", "B", "A", "B", "A"))

# Applying the mean function over subsets defined by the categories
tapply(values, categories, mean)
```

In this example, `tapply()` calculates the mean of the `values` vector for each category defined by the `categories` factor. It splits the `values` vector into subsets based on the categories and applies the `mean()` function to each subset, returning the mean values for categories "A" and "B".

The result would be something like:

```
A    B
30   30
```

This indicates that the mean value for category "A" is 30, and the mean value for category "B" is also 30 in this case.

`lapply()`, `sapply()`, and similar functions work on elements of lists or vectors. In contrast, `tapply()` focuses on splitting a vector by factors and applying a function to these subsets, providing aggregated results for each subset determined by the factors.

Diffrences between `apply` in base R and the package `dplyr`:

1. **`apply` functions in base R:**

   - The `apply` family of functions (`apply()`, `lapply()`, `sapply()`, `vapply()`, etc.) in base R are used for applying a function over margins of arrays or data structures like matrices, arrays, and lists.
   - `apply()` is used primarily for applying functions to the rows or columns of matrices or arrays, while `lapply()` and `sapply()` are more focused on lists.
   - These functions are useful for repetitive operations across rows or columns without explicitly using loops.

   Example:

   ```r
   # Creating a matrix
   mat = matrix(1:12, nrow = 3, ncol = 4)

   # Applying sum function to rows (1) or columns (2) of the matrix
   apply(mat, 1, sum)  # Sums of each row
   apply(mat, 2, sum)  # Sums of each column
   ```

2. **`dplyr` package:**

   - `dplyr` is a powerful package in R for data manipulation and transformation. It provides a set of functions (`filter()`, `mutate()`, `select()`, `group_by()`, `summarize()`, etc.) that enable easy and intuitive data manipulation.
   - It's designed to work well with data frames and offers a more streamlined and readable syntax for performing common data manipulation tasks.

   Example:

   ```r
   library(dplyr)

   # Creating a sample data frame
   df = data.frame(
     Name = c("Alice", "Bob", "Charlie"),
     Age = c(25, 30, 28),
     Salary = c(40000, 50000, 45000)
   )

   # Filtering and selecting specific rows and columns
   filtered_df = df %>%
     filter(Age > 25) %>%
     select(Name, Salary)

   filtered_df
   ```

   This `dplyr` example filters rows where `Age` is greater than 25 and selects only the `Name` and `Salary` columns. The `%>%` operator (pipe) chains together multiple operations, making the code more readable and concise.

In summary, the `apply` family in base R is ideal for applying functions to matrices, arrays, or lists across rows or columns, while `dplyr` focuses on intuitive data manipulation operations for data frames, providing a cleaner syntax and ease of use for common data transformation tasks.

More on iteration: `purrr` package

About `purrr` (from official website https://purrr.tidyverse.org)

`purrr` enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors. If you've never heard of FP before, the best place to start is the family of `map()`functions which allow you to replace many for loops with code that is both more succinct and easier to read. The best place to learn about the `map()` functions is the iteration chapter in R for data science.

Usage

The following example uses `purrr`to solve a fairly realistic problem: split a data frame into pieces, fit a model to each piece, compute the summary, then extract the R^2.

```
library(purrr)

mtcars |>
  split(mtcars$cyl) |>  # from base R
  map(\(df) lm(mpg ~ wt, data = df)) |>
  map(summary) %>%
  map_dbl("r.squared")
```

```
#>         4         6         8
#> 0.5086326 0.4645102 0.4229655
```

This example illustrates some of the advantages of `purrr` functions over the equivalents in base R:

- The first argument is always the data, so purrr works naturally with the pipe.
- All `purrr` functions are type-stable. They always return the advertised output type (`map()` returns lists; `map_dbl()` returns double vectors), or they throw an error.
- All `map()` functions accept functions (named, anonymous, and lambda), character vector (used to extract components by name), or numeric vectors (used to extract by position).

Detailed Usage

`purrr` is a powerful package in R that focuses on enhancing and simplifying the process of working with functions and vectors. Developed as part of the tidyverse ecosystem, `purrr` provides a consistent and coherent set of tools for functional programming, iteration, and working with lists and vectors.

Here are some key aspects and functionalities of `purrr`:

1. **Functional Programming:**
   - `purrr` promotes functional programming paradigms in R, enabling users to work with functions as first-class objects.
   - It provides functions like `map()`, `map2()`, `pmap()`, `walk()`, and more, which allow applying functions over elements of lists or vectors.
2. **Consistency Across Data Structures:**
   - `purrr` functions exhibit consistent behavior across various data structures, such as lists, vectors, and data frames.
   - These functions can work seamlessly with different data structures, making code more readable and maintainable.

3. **Iteration and Mapping:**
    - `map()` is a key function in `purrr` that iterates over elements of a list or vector, applying a function to each element and returning the results.
    - `map2()` is similar to `map()` but allows iterating over two vectors simultaneously.
    - `pmap()` extends this functionality to iterate over multiple vectors or lists simultaneously.
4. **Simplified and Cleaner Syntax:**
    - `purrr` functions often provide a more consistent and cleaner syntax compared to base R functions for similar operations.
    - The use of the pipe `%>%` from the tidyverse allows chaining `purrr` functions together, resulting in more readable code.
5. **Working with Lists and Data Frames:**
    - `purrr` provides functions to efficiently manipulate and iterate over elements in lists and data frames.
    - Functions like `map()` and `map_dbl()` can be used to apply functions to each column of a data frame and collect results in an output structure.

Example of `map()` in `purrr`:

```
library(purrr)

# Applying sqrt function to each element of a list
numbers = list(a = 1:5, b = 6:10)
result = map(numbers, sqrt)

result
# Output: List of 2
# $ a: num [1:5] 1 1.41 1.73 2 2.24
# $ b: num [1:5] 2.45 2.65 2.83 3 3.16
```

This code applies the `sqrt()` function to each element of the list `numbers`, returning a list with the square roots of each element.

`purrr` simplifies and enhances functional programming in R, offering a consistent and expressive way to work with functions, lists, vectors, and data frames, making data manipulation and iteration more straightforward and concise.

Examples

Here are some specific functionalities and examples of `purrr`:

1. **Mapping Functions:**

   `map()` applies a function to each element of a list or vector, returning a list.

   ```
   library(purrr)

   # Squaring each element in a vector using map()
   numbers = 1:5
   squared = map(numbers, ~ .x^2)

   squared
   # Output: List of 5
   ```

```
# $ : int 1
# $ : int 4
# $ : int 9
# $ : int 16
# $ : int 25
```

2. **Mapping Functions over Multiple Inputs:**

   `map2()` allows applying a function that takes two inputs to corresponding elements of two vectors.

```
# Multiplying elements of two vectors element-wise
vector1 = 1:5
vector2 = 6:10
product = map2(vector1, vector2, ~ .x * .y)

product
# Output: List of 5
# $ : int 6
# $ : int 14
# $ : int 24
# $ : int 36
# $ : int 50
```

3. **Working with Data Frames:**

   `map_df()` and similar functions allow applying a function to each column of a data frame and combining results into a data frame.

```
# Creating a data frame
df = data.frame(A = 1:5, B = 6:10)

# Doubling each column in the data frame
doubled = map_df(df, ~ .x * 2)

doubled
# Output: A tibble: 5 × 2
#       A     B
#   <dbl> <dbl>
# 1     2    12
# 2     4    14
# 3     6    16
# 4     8    18
# 5    10    20
```

4. **Iteration and Applying Functions:**

   `walk()` applies a function to each element without returning a result, useful for side effects or performing operations without output.

```
# Printing each element of a list using walk()
fruits = list("apple", "banana", "orange")
walk(fruits, print)
# Output:
# [1] "apple"
# [1] "banana"
# [1] "orange"
```

`purrr` simplifies functional programming by providing intuitive functions (`map()`, `map2()`, `walk()`, etc.) that allow iteration over elements, applying functions, and collecting results, making code more concise and readable in scenarios involving lists, vectors, and data frames.

(in the slide) Function `reduce()` and `accumulate()`

Both `reduce()` and `accumulate()` are powerful functions from the `purrr` package that facilitate iterative calculations over a sequence, accumulating or reducing values based on a specified function.

Here's an explanation of each:

1. **`reduce()` Function:**

   - `reduce()` is used to successively apply a function to pairs of elements in a sequence, reducing it to a single value.
   - The function provided to `reduce()` should take two arguments and return a single value.
   - It starts by applying the function to the first two elements, then uses the result along with the next element, and so on, until the sequence is exhausted.

   Example of `reduce()`:

   ```r
   library(purrr)

   # Summing all elements in a vector using reduce()
   numbers = 1:5
   total_sum = reduce(numbers, `+`)

   total_sum
   # Output: 15 (1 + 2 + 3 + 4 + 5 = 15)
   ```

   Here, `reduce()` adds all the elements in the `numbers` vector by applying the addition function (`+`) iteratively.

2. **`accumulate()` Function:**

   - `accumulate()` is similar to `reduce()` but returns a sequence of intermediate results instead of a single value.
   - It applies a function cumulatively to the sequence and returns a vector of values, representing the intermediate results at each step.

   Example of `accumulate()`:

   ```r
   # Calculating cumulative product of elements in a vector using accumulate()
   factors = c(2, 3, 4, 5)
   cumulative_product = accumulate(factors, `*`)

   cumulative_product
   # Output: 2 6 24 120 (2, 2*3, 2*3*4, 2*3*4*5)
   ```

   Here, `accumulate()` applies the multiplication function (`*`) to each element in `factors`, returning a vector of cumulative products at each step.

`reduce()` aggregates a sequence into a single value based on a function, while `accumulate()` returns a sequence of intermediate results. Both functions are helpful for iterative calculations and provide different ways to process sequences of values in R.

Parallel Computing

Related Packages

- `parallel` package: displays the number of CPU cores to assign all or part to a task.
- `foreach` package: provides `%do%` and `%dopar%` operators to submit tasks for sequential or parallel computation
- `` ` iterators `` package: splits `data.frame, tibble, matrix` into rows/columns for submitting parallel tasks.

Step-by-step Guidance

1. **Prepare Data:**

   Assume you have a large data frame named `my_data` that you want to process in parallel.

2. **Setup Parallel Processing:**

   Load necessary packages and initialize parallel processing capabilities.

   ```
   library(parallel)
   library(doParallel)
   library(foreach)
   library(iterators)

   # Set the number of cores/processors to be used
   num_cores = detectCores()

   # Initialize parallel backend
   cl = makeCluster(num_cores)
   registerDoParallel(cl)
   ```

3. **Split Data Frame into Chunks:**

   Use the `iter()` function from the `iterators` package to create an iterator for chunks of your data frame.

   ```
   # Define chunk size
   chunk_size = nrow(my_data) / num_cores

   # Create an iterator for the chunks
   my_iterator = iter(my_data, by = "row", chunksize = chunk_size)
   ```

4. **Perform Parallel Computation:**

   Use `foreach()` from the `foreach` package along with `%dopar%` to apply a function to each chunk in parallel.

   ```
   # Define a function to process each chunk
   process_chunk = function(chunk) {
     # Your processing logic for each chunk goes here
     # For example: summary(chunk)
     # Replace summary() with your specific data processing task
   }

   # Apply the function to each chunk in parallel
   results = foreach(chunk = my_iterator, .combine = rbind) %dopar% {
     process_chunk(chunk)
   }
   ```

5. **Combine Results:**

   Collect and combine the results obtained from parallel processing.

```r
# Combine or process the results obtained from parallel computation
final_result = do.call(rbind, results)

# Close the parallel cluster
stopCluster(cl)
```

Replace the `process_chunk()` function with your specific data processing task. This approach parallelizes the processing of chunks of the data frame across multiple cores, allowing for faster computations, especially with large datasets.

Note:

- When the task is completed, the allocated CPU core is reclaimed.
- Ensure that your specific data processing task is compatible with parallelization and that the benefits of parallel computing outweigh the overhead of parallelization.
- Also, consider potential dependencies or shared resources among iterations when parallelizing computations.

(in the slide) Function `foreach()`

Simple usage

Description `%do%` and `%dopar%` are binary operators that operate on a `foreach` object and an `R` expression. The expression, `ex`, is evaluated multiple times in an environment that is created by the `foreach` object, and that environment is modified for each evaluation as specified by the `foreach` object. `%do%` evaluates the expression sequentially, while `%dopar%` evaluates it in parallel. The results of evaluating `ex` are returned as a list by default, but this can be modified by means of the `.combine` argument.

Usage

```r
foreach(
  ...,
  .combine,
  .init,
  .final = NULL,
  .inorder = TRUE,
  .multicombine = FALSE,
  .maxcombine = if (.multicombine) 100 else 2,
  .errorhandling = c("stop", "remove", "pass"),
  .packages = NULL,
  .export = NULL,
  .noexport = NULL,
  .verbose = FALSE
)
e1 %:% e2
when(cond)
obj %do% ex
obj %dopar% ex
times(n)
```

Nested `foreach`

Expanded knowledge, not featured on slide, for understanding only.

Nested `foreach` loops in R allow for the iteration over multiple levels of nested structures or combinations of iterators. This approach is particularly useful when dealing with hierarchical data or when you need to perform computations on multiple levels of nested objects simultaneously.

Here's an overview of nested `foreach` loops:

1. **Nested Iteration:**

   `foreach` supports nesting, allowing you to iterate over multiple levels of nested structures, such as lists within lists or matrices within lists.

   ```
   library(foreach)

   # Example: Nested foreach loop iterating over a list of lists
   outer_list = list(list(a = 1, b = 2), list(c = 3, d = 4))

   foreach(inner_list = outer_list) %:% {
     foreach(element = inner_list) %do% {
       # Process each element within the nested structure
       print(element)
     }
   }
   ```

   This code iterates over each element of `outer_list`, which contains inner lists. Within each inner list, it iterates over the elements.

2. **Combining Iterators:**

   You can combine different iterators using `%:%` to create nested iterations.

   ```
   # Example: Nested foreach loop with combined iterators
   values = 1:3
   letters = letters[1:4]

   foreach(i = values) %:% foreach(letter = letters) %do% {
     # Perform operations using both iterators
     print(paste("Value:", i, "| Letter:", letter))
   }
   ```

   This code creates nested iterations, iterating over `values` and `letters` simultaneously.

3. **Applying Nested Functions:**

   Nested `foreach` loops are valuable when applying functions or performing operations that require iterating over multiple levels of nested data structures or combinations.

   ```
   # Example: Applying a function with nested foreach loops
   matrix_list = list(matrix(1:4, nrow = 2), matrix(5:8, nrow = 2))

   foreach(mat = matrix_list) %:% foreach(element = as.vector(mat)) %do% {
     # Perform computations on each element of each matrix
     print(element * 2)
   }
   ```

   Here, it iterates over a list of matrices and then iterates over each element within the matrices to perform computations.

Nested `foreach` loops in R allow for flexible and efficient iterations over hierarchical or nested structures, enabling complex computations, data manipulations, or simulations involving multiple levels of nested objects or iterators.