Syllabus :: Module -2 (Reading and writing data- Apply Level)

Importing data from Text files and other software, Exporting data, importing data from databases- Database Connection packages, Missing Data - NA, NULL

Combining data sets, Transformations, Binning Data, Subsets, summarizing functions.

Data Cleaning, Finding and removing Duplicates, Sorting.

Overview of Textbook Contents Aligned with the Syllabus [Module 2]

- **Importing data from Text files and other software, Exporting data**
    - This topic covers how to **bring data into R from external sources like text files and other software formats**.
    - "R in a Nutshell" discusses **reading data from text files** in the context of importing data. It also mentions **exporting data**.
    - The syllabus also mentions **exporting data** as part of this topic.
- **Importing data from databases- Database Connection packages**
    - This section focuses on **retrieving data stored in databases using specific R packages**.
    - "R in a Nutshell" has a section on **importing data from databases**, specifically mentioning **Database Connection packages** such as **DBI**, **RODBC**, and **TSBDI**.
- **Missing Data - NA, NULL**
    - This topic deals with how R represents and handles **missing values**, specifically the special values **NA** (Not Available) and **NULL**.
    - "R in a Nutshell" explicitly discusses **special values** including **NA**. While it doesn't dedicate a separate section to `NULL` in the main data handling chapters, it does mention `NULL` as a special object type. The `predict` function's `na.action` argument and the `lm` function's handling of `NA` values in data are also mentioned.
- **Combining data sets, Transformations**

    - This section covers **merging or joining multiple datasets** together and **transforming variables** within a dataset.
    - "R in a Nutshell" addresses **combining data sets** and includes a section on **transformations**. This section details methods for **reassigning variables** and using the `transform` **function**. It also covers **pasting together** data.
- **Binning Data, Subsets, summarizing functions**
    - This topic includes **categorizing continuous data into bins**, **selecting specific portions (subsets) of a dataset**, and using **functions to get summary information about the data**.
    - While "R in a Nutshell" doesn't have a section explicitly titled "Binning Data," it does discuss techniques like creating **factor vectors** which can be related to binning. It has a section on **subsets** as part of data manipulation. The syllabus itself later mentions **summarizing functions** again. "R in a Nutshell" has a section on **summarizing functions** such as `summary`, `tapply`, and `aggregate`.
- **Data Cleaning, Finding and removing Duplicates, Sorting**
    - This covers the process of **identifying and correcting errors or inconsistencies in data (data cleaning)**, specifically **finding and removing duplicate records**, and **arranging data in a specific order (sorting)**.
    - "R in a Nutshell" includes a section specifically on **Data Cleaning**. This section further discusses **finding and removing duplicates** and **sorting** data.

Importing data from Text files and other software, Exporting data

---

Importing Data from Text Files

R offers a family of functions based on the `read.table` function for importing delimited text files. Some of the most commonly used functions include:

- **`read.csv`**: Specifically designed for comma-separated value (CSV) files. It assumes a header row and uses a comma as the default separator.
- **`read.delim`**: Used for files where fields are delimited by a specific character, with the tab character (`\t`) being the default.
- **`read.csv2` and `read.delim2`**: These are variations that use a semicolon (`;`) as the separator and a comma (`,`) for the decimal point, which is common in some European locales.

These functions have various arguments to control how the data is read. Some important ones include:

- `sep`: Specifies the delimiter character.
- `header`: A logical value indicating if the first row of the file contains variable names.
- `quote`: The set of quoting characters.
- `dec`: The character used for decimal points.
- `fill`: A logical value indicating whether to pad rows with differing numbers of columns.
- `comment.char`: Character string indicating comments; lines beginning with it are ignored.
- `na.strings`: Character vector of strings which are to be interpreted as `NA` values.
- `stringsAsFactors`: A logical value indicating whether character columns should be converted to factors.
- `nrows`: The number of rows to read from the file. This can be useful for testing with large files.

You can even **import data directly from a URL** by providing the URL as the `file` argument to these functions. For example, you can fetch a CSV file from a website.

For **fixed-width files**, where variables are differentiated by their location on each line, R provides the `read.fwf` function. This function requires specifying the widths of each field.

For more complex scenarios, you can read data into R **one line at a time** using the `readLines` function. Another versatile function is `scan`, which allows reading data in various formats and specifying the data type to be read.

"R in a Nutshell" also provides tips for creating text files that are easy to import into R. For example, when exporting from Microsoft Excel, you can choose between CSV and tab-delimited files and should ideally specify Unix-style line delimiters. When exporting from databases, using GUI tools to export the results can be easier than command-line scripts. For very large text files, the book suggests using scripting languages like Perl, Python, or Ruby to preprocess the data into a more manageable format before importing it into R.

Importing Data from Other Software

While many software packages can export data as text files, R can also directly read files in various other formats using functions often provided by **additional packages**. Table 11-1 in "R in a Nutshell" lists some of these functions:

- **ARFF (Weka)**: `read.arff` (in `foreign` package) and `write.arff`.
- **DBF (dBase)**: `read.dbf` (in `foreign` package) and `write.dbf`.
- **Stata**: `read.dta` (in `foreign` package) and `write.dta`.
- **Epi Info**: `read.epiinfo` (in `foreign` package).
- **Minitab**: `read.mtp` (in `foreign` package).
- **Octave**: `read.octave` (in `foreign` package).

- **S (older versions)**: `read.S` (in `foreign` package).
- **SPSS**: `read.spss` (in `foreign` package).
- **SAS Permanent Dataset**: `read.ssd` (in `foreign` package).
- **Systat**: `read.systat` (in `foreign` package).
- **SAS XPORT File**: `read.xport` (in `foreign` package).

The **`foreign` package** is a crucial package that provides many of these functions for reading data files from other statistical software. You would typically need to install and load this package using `install.packages("foreign")` and `library(foreign)` before using these functions.

Additionally, for users familiar with Microsoft Excel, the **`RExcel` software** allows running R directly from within Excel on Microsoft Windows systems, facilitating data exchange. Information about this software can be found at the provided URL.

For importing data from **databases**, R provides **Database Connection Packages** like `RODBC` and `DBI`. The syllabus explicitly mentions these. These packages allow you to connect to various database systems and retrieve data directly into R. This topic is covered in detail in a subsequent section of Module 2.

Exporting Data

R can export R data objects, primarily data frames and matrices, to text files using the **`write.table` function**. This function has several arguments to control the output format:

- **`x`**: The data object to be exported.
- **`file`**: The name of the file or a connection object to write to.
- **`append`**: A logical value indicating whether to append to an existing file or overwrite it.
- **`quote`**: A logical value indicating whether to quote character or factor values.
- **`sep`**: The character used to separate values within a row (e.g., ",", "\t").
- **`eol`**: The character to append at the end of each line (default is "\n").
- **`na`**: The string to use for representing `NA` values (default is "NA").
- **`dec`**: The character used for the decimal separator.
- **`row.names`**: A logical value indicating whether to include row names in the output.
- **`col.names`**: A logical value indicating whether to include column names.
- **`qmethod`**: Specifies how to handle quotes within quoted fields ("escape" or "double").

You can also export R data objects (like individual variables, lists, or entire workspaces) in a binary format using the **`save` function**. This saves the objects to a file (typically with the extension `.RData` or `.rda`) that can be reloaded into R later using the **`load` function**. The `save` function allows you to specify which objects to save and the `file` name. Connections, such as those created with `gzfile` for compressed files, can be used with `save` and `load`.

In summary, R provides a robust set of tools for importing data from various text file formats and other software, often relying on the base functions for text files and additional packages like `foreign` for other formats. Similarly, the `write.table` and `save` functions enable flexible exporting of data and R objects. Understanding these functions and their arguments is crucial for effectively managing data within R.

Importing data from databases - Database Connection packages

There are primarily two approaches to getting data from databases into R: **exporting the data to a text file and then importing it into R**, or **establishing a direct connection from R to the database using database connection packages**.

Export Then Import

One effective method, particularly for large datasets (1 GB or more) accessed once for analysis, is to first **export the data from the database to a text file** (as discussed in the "Importing data from Text files and other software" topic) and then import that file into R. "R in a Nutshell" suggests that importing from text files can be significantly faster than direct database connections for very large datasets. For guidance on importing these exported text files, refer to the "Text Files" section. Tools for querying and exporting data from databases often include GUI options like Toad for Data Analysts (for various databases), MySQL Query Browser, and Oracle SQL Developer, which can simplify the export process compared to command-line scripts.

Database Connection Packages

For scenarios requiring regular reports or repeated analysis, directly importing data into R via database connections can be more efficient. To achieve this, R utilizes **optional packages** that need to be installed based on the specific database(s) you intend to connect to and the preferred connection method. "R in a Nutshell" highlights two main sets of database interfaces available in R:

- **RODBC (R Open DataBase Connectivity)**: This package allows R to retrieve data from databases using **ODBC (Open DataBase Connectivity)** connections. ODBC is a standard interface that enables different programs to connect to various databases.
    - To use RODBC, you generally need to:
        1. **Install the RODBC package** in R using `install.packages("RODBC")`.
        2. If necessary, **install the ODBC drivers** for your specific database platform (e.g., MySQL, Oracle, PostgreSQL, Microsoft SQL Server, SQLite). Table 11-2 in "R in a Nutshell" provides a list of some sources for ODBC drivers.

3. **Configure an ODBC connection** to your database. This typically involves setting up a **Data Source Name (DSN)**. Examples for configuring an ODBC connection to an SQLite database are provided for both **Microsoft Windows** and **Mac OS X**.

- Once configured, you can use functions from the RODBC package to interact with the database:
    1. **`odbcConnect(dsn, uid = "", pwd = "", ...)`**: Establishes a connection to the database using the specified DSN and optional username (`uid`) and password (`pwd`). It returns an object of class `RODBC` representing the connection (often called a channel).
    2. **`odbcGetInfo(channel)`**: Retrieves information about the ODBC connection.
    3. **`sqlTables(channel)`**: Lists the tables available in the database for the connected user.
    4. **`sqlColumns(channel, sqtable)`**: Provides detailed information about the columns in a specific table (`sqtable`).
    5. **`sqlPrimaryKeys(channel, sqtable)`**: Discovers the primary keys for a table.
    6. **`sqlFetch(channel, sqtable, ..., colnames = , rownames = )`**: Fetches the entire content of a table or view (`sqtable`) into an R data frame.
    7. **`sqlQuery(channel, query, errors = TRUE, max = 0, ..., rows_at_time = )`**: Executes an arbitrary SQL query in the database and returns the results as a data frame. The `max` argument can be used to fetch results piecewise for very large queries, with `sqlGetResults` used to retrieve subsequent rows.
    8. **`sqlSave(channel, data, tablename, ...)`**, **`sqlUpdate(channel, data, tablename, ...)`**: Functions for writing an R data frame to a database table or updating an existing table (see help files for details).
    9. **`odbcClose(channel)`**: Closes the database connection.
    10. **`odbcCloseAll()`**: Closes all open RODBC channels.
- RODBC also has a second set of lower-level functions like `odbcQuery`, `odbcTables`, `odbcColumns`, `odbcPrimaryKeys` (for executing queries but not fetching results) and `odbcFetchResults` (to get the results), but the higher-level functions are generally more convenient.

- **DBI (Database Interface)**: DBI is not a single package but rather a **framework and a set of packages** for accessing databases. It provides a common database abstraction for R software. To connect to specific databases using DBI, you need to install **additional driver packages**. Table 11-3 in "R in a Nutshell" lists some of these packages and the corresponding databases:

    - **RMySQL** for MySQL
    - **RSQLite** for SQLite
    - **ROracle** for Oracle
    - **RPostgreSQL** for PostgreSQL
    - **RJDBC** for any database with a JDBC driver
    - To use DBI:
        1. **Install the appropriate driver package** (e.g., `install.packages("RSQLite")`). Installing a driver package often automatically loads the DBI package as well.
        2. **Open a connection** using the `dbConnect(drv, ...)` function. The `drv` argument can be a driver object created with `dbDriver("driverName")` (e.g., `dbDriver("SQLite")`) or a character string specifying the driver. Additional arguments, specific to the database type (like `dbname` for SQLite), are also required.
        3. **Get database information** using functions like `dbGetInfo(con)` (for connection details), `dbListConnections(drv)` (for connections associated with a driver), `dbListTables(con)` (for available tables), and `dbListFields(con, "tableName")` (for column names).
        4. **Query the database** using functions like `dbGetQuery(con, sqlStatement)` which executes an SQL query and returns the result as a data frame. Alternatively, you can use `dbSendQuery(con, sqlStatement)` to send a query and then `fetch(res)` to retrieve the results, allowing for piecewise fetching.
        5. **Read or write entire tables** using `dbReadTable(con, "tableName")` and `dbWriteTable(con, dataFrame, "tableName", ...)` respectively. You can also check if a table exists with `dbExistsTable(con, "tableName")` and remove it with `dbRemoveTable(con, "tableName")`.
        6. **Close the connection** using `dbDisconnect(con)`. You can also unload the driver with `dbUnloadDriver(drv)` to free system resources.
    - DBI utilizes **S4 objects** to represent drivers and connections, which can help in writing better code.
- **TSDBI (Time Series Database Interface):** This is another database interface in R specifically designed for **time series data**. TSDBI packages are available for various databases, including MySQL (TSMySQL), SQLite (TSSQLite), Fame (TSFame), PostgreSQL (TSPostgreSQL), and any database with an ODBC driver (TSODBC).

Choosing between RODBC and DBI often depends on factors like driver availability across different operating systems, the need for specific database features or performance, and personal preference regarding coding style (RODBC is more procedural, while DBI uses S4 objects).

Finally, for working with data from **Hadoop**, "R in a Nutshell" mentions the **RHadoop package** as a mature and well-integrated project, which includes functions for moving data in and out of Hadoop clusters and executing Map/Reduce jobs. Another option mentioned is using Hadoop streaming with other applications like R. The book refers to the "R and Hadoop" section for more details.

Missing Data - NA and NULL

NA (Not Available)

In R, **NA** values are used to represent **missing values**. **NA** stands for "not available". You might encounter **NA** values in several situations:

- When **text is loaded into R**, **NA** can be used to represent missing values in the original data.
- When **data is loaded from databases**, **NULL values in the database are often replaced by NA in R**.

If you **expand the size of a vector**, matrix, or array beyond the initially defined size, the newly created spaces will be filled with the value **NA**. For example:

```
> v <- c(1,2,3)

> length(v) <- 4

> v

 1  2  3 NA
```

- When the length of the vector `v` is increased to 4, the fourth element is assigned the value **NA**. Similarly, expanding a vector `w` also results in **NA** values for the uninitialized elements.

It's important to note that **NA** is a specific value used to indicate that a piece of data is missing or not applicable. Many functions in R have an argument, often `na.rm` (NA remove), which specifies how **NA** values should be treated during calculations. By default, if any value in a vector is **NA**, functions like `mean()` will often return **NA**. However, setting `na.rm=TRUE` will instruct the function to ignore the **NA** values in the calculation.

Furthermore, many modeling functions in R have an `na.action` argument to specify how **NA** values in the data should be handled during model fitting. Common options include `na.omit` (remove rows with NA values) and `na.fail` (return an error if NA values are present).

NULL

NULL is another special object in R, represented by the symbol `NULL`. The symbol `NULL` always points to the same object. **NULL** is distinct from **NA**, `Inf`, `-Inf`, and `NaN`.

NULL is often used in the following ways:

- As an **argument in functions** to indicate that no value was assigned to that argument.
- As a **return value from some functions** to signify an absence of a result or a specific condition.

NULL represents the absence of an object. It has a length of zero and no type.

Differences between NA and NULL

A key distinction is that **NA** represents a missing value within a data structure (like a vector or data frame), whereas **NULL** represents the absence of a data structure or object itself. An **NA** in a vector makes the length of the vector remain the same, but indicates that a particular element's value is unknown. **NULL**, on the other hand, effectively means "nothing is there."

For example, if you create a list and assign **NULL** to one of its elements, that element will effectively disappear, and the length of the list might change depending on how it's handled. If you assign **NA**, the element will still exist but have a missing value.

In summary, both **NA** and **NULL** are important concepts for handling data in R. **NA** signifies missing values within a dataset, which is common in real-world data. **NULL** represents the absence of an object and is often used in programming contexts like function arguments and return values. Understanding their differences is crucial for effective data manipulation and analysis in R.

Combining data sets , Transformations

Combining Datasets in R

There are several functions in R that allow you to combine multiple data structures into a single one:

`paste()`: This function is used to **concatenate multiple character vectors into a single character vector**. If you try to concatenate a vector of another type, it will be coerced to a character vector first. By default, the values being pasted together are separated by a space, but you can specify a different separator using the `sep` argument or no separator at all. The `collapse` argument can be used to concatenate all the values in the resulting vector into a single character string with a specified separator.

```
> x <- c("a", "b", "c")
```

```
> y <- c("1", "2", "3")
> paste(x, y)

"a 1" "b 2" "c 3"

> paste(x, y, sep="-")

"a-1" "b-2" "c-3"

> paste(x, y, sep="-", collapse="#")

"a-1#b-2#c-3"
```

`cbind()`: This function combines objects (like vectors, matrices, or data frames) by **adding columns**. You can think of this as combining tables horizontally. For this to work effectively, the objects you are combining should generally have the same number of rows. "R in a Nutshell" provides an example of combining a data frame `top.5.salaries` with another data frame `more.cols` using `cbind()`.

```
 > top.5.salaries <- data.frame(name.last=c("Manning", "Brady"), name.first=c("Peyton", "Tom"), salary=c(18700000, 14626720))

> more.cols <- data.frame(year=c(2008, 2008), rank=c(1, 2))

> cbind(top.5.salaries, more.cols)

  name.last name.first   salary year rank

1   Manning     Peyton 18700000 2008    1

2     Brady        Tom 14626720 2008    2
```

`rbind()`: This function combines objects (like vectors, matrices, or data frames) by **adding rows**. This can be visualized as stacking tables vertically. For `rbind()` to work properly, the objects should have the same number of columns and ideally the same column names. "R in a Nutshell" demonstrates combining two data frames, `top.5.salaries` and `next.three`, using `rbind()`.

```
 > top.5.salaries <- data.frame(name=c("A", "B"), value=c(10, 20))

> next.three <- data.frame(name=c("C", "D"), value=c(30, 40))

> rbind(top.5.salaries, next.three)

  name value

1    A    10

2    B    20
```

```
3   C   30

4   D   40
```

`merge()`: This is a powerful function for combining two data frames based on **common columns or row names**. The `merge()` function in R is similar to database join operations. You can specify the common columns to join by using the `by`, `by.x`, and `by.y` arguments. The `all`, `all.x`, and `all.y` arguments control whether rows with no match in the other data frame should be included (similar to OUTER, LEFT OUTER, and RIGHT OUTER joins in SQL). By default, `merge()` performs a natural join (only rows with matching values in the specified columns are included).

```
 > df1 <- data.frame(ID=c(1, 2, 3), Name=c("X", "Y", "Z"))

> df2 <- data.frame(ID=c(2, 3, 4), Value=c(100, 200, 300))

> merge(df1, df2, by="ID") # Inner join by default

  ID Name Value

1  2    Y   100

2  3    Z   200

> merge(df1, df2, by="ID", all=TRUE) # Full outer join

  ID Name Value

1  1    X    NA

2  2    Y   100

3  3    Z   200

4  4 <NA>   300
```

`make.groups()` (from the `lattice` package): Sometimes you might want to combine a set of similar objects (either vectors or data frames) into a single data frame, with an additional column that labels the source of each object. The `make.groups()` function in the `lattice` package is useful for this purpose.

```
 > library(lattice)

> vec1 <- 1:3

> vec2 <- 4:6
> make.groups(Vector1 = vec1, Vector2 = vec2)

  data   which

1    1  Vector1

2    2  Vector1

3    3  Vector1

4    4  Vector2

5    5  Vector2

6    6  Vector2
```

These functions provide various ways to combine datasets in R depending on the structure of your data and the desired outcome. Chapter 12 of "R in a Nutshell" provides more detailed explanations and examples of these and other data preparation techniques.

Transformations in R

Chapter 12 of "R in a Nutshell" also covers data transformations, which involve modifying variables within a data set.

- **Reassigning Variables:**
  - You can create new variables or modify existing ones within a data frame using standard assignment. For instance, `dow30$mid <- (dow30$High + dow30$Low) / 2` creates a new column named `mid` in the `dow30` data frame.
- **The `transform()` Function:**
  - The `transform()` function provides a convenient way to change variables in a data frame. Its formal definition is `transform(_data, ...)`.
  - You specify the data frame as the first argument, followed by a series of expressions that use variables from that data frame. `transform()` applies each expression and returns the modified data frame.

For example:

```
 > dow30.transformed <- transform(dow30, Date=as.Date(Date),

+   mid = (High + Low) / 2)
```

- **Applying a Function to Each Element of an Object:**
  - R offers several functions to apply a function to a set of objects, including:
    - `lapply()`: Applies a function to each element of a list and returns a list.

        - `sapply()`: A user-friendly version of `lapply()` that tries to simplify the output to a vector or matrix.
        - `apply()`: Applies a function to the margins of an array or matrix.
        - `tapply()`: Applies a function to each cell of a ragged array, based on combinations of factor levels.
        - `mapply()`: A multivariate version of `lapply()` that applies a function to corresponding elements of multiple vectors, lists, or other vector-like objects.
- **The `plyr` Package:**
    - The `plyr` package provides a set of functions (with names like `*dply`) for applying a function to an R data object (array, data frame, or list) and returning the results in various formats.
- **Binning Data:**
    - Binning involves grouping observations into bins based on the value of a variable.
    - **Shingles:** These represent intervals and are used extensively in the `lattice` package for conditioning variables. The `shingle()` function handles shingles.
    - **`cut()` Function:** This function divides the range of a numeric variable into intervals (bins) and codes the values according to which interval they fall into.
- **Reshaping Data:**
    - Reshaping transforms the structure of data, often between "wide" and "long" formats.
    - `stack()` and `unstack()` are basic functions for this.
    - The `reshape()` function is more powerful but can be complex. The `direction` argument specifies whether to transform to "long" or "wide" format.
    - The `reshape2` package offers a more intuitive approach with two main functions:
        - `melt()`: Turns a data frame into a long format.
        - `cast()`: Takes a melted data frame and reshapes it into a different format.

These methods provide a comprehensive set of tools for combining and transforming data sets in R, allowing you to prepare your data effectively for analysis.

Binning Data [Note : It is a part of Data Transformation Techniques]

Binning data is a common and important **data transformation** technique in R. It involves grouping a set of observations into **bins** or intervals based on the value of a specific variable. This is often done to summarize data, to make patterns more apparent, or to prepare data for certain types of analysis or modeling.

Here are the key methods for binning data in R as described in the sources:

1. The `cut()` Function:

- The `cut()` function is specifically designed for taking a **continuous variable and splitting it into discrete pieces (bins)**.
- Its default form for numeric vectors is `cut(x, breaks, labels = NULL, include.lowest = FALSE, right = TRUE, dig.lab = 3, ordered_results = FALSE)`.
    - `x`: This is the **numeric vector** that you want to convert into a factor by binning its values.
    - `breaks`: This argument specifies **how the bins should be defined**. It can be either:
        - A **single integer value** indicating the desired number of breakpoint intervals. `cut()` will then try to create approximately equal-width intervals.
        - A **numeric vector** specifying the actual breakpoint values that define the boundaries of the bins.
    - `labels`: This is an optional argument that allows you to provide **labels for the levels (bins) in the output factor**. If not provided, labels are generated based on the break points.
    - `include.lowest`: A **logical value** indicating whether the interval should include the lowest breakpoint if `right = TRUE`, or the highest if `right = FALSE`. The default is `FALSE`.
    - `right`: A **logical value** that specifies whether the intervals should be **closed on the right and open on the left** (i.e., `(a, b]`). If `right = FALSE`, intervals will be open on the right and closed on the left (i.e., `[a, b)`). The default is `TRUE`.
    - `dig.lab`: An integer that controls the **number of digits used when generating labels** if `labels` are not explicitly specified. The default is `3`.
    - `ordered_results`: A **logical value** indicating whether the resulting factor should be an **ordered factor**. The default is `FALSE`.

An example in ["R in a Nutshell 2nd edition.pdf"] demonstrates how to use `cut()` to bin batting averages and then use `table()` to count the number of players in each bin:

```
> library(nutshell)

> data(batting.2008)

> batting.2008.AB <- transform(batting.2008, AVG = H/AB)

> batting.2008.over100AB <- subset(batting.2008.AB, subset=(AB > 100))
> battingavg.2008.bins <- cut(batting.2008.over100AB$AVG,breaks=10)

> table(battingavg.2008.bins)
```

- This code first calculates the batting average (`AVG`), then subsets the data to include players with over 100 at-bats, and finally uses `cut()` to divide the batting averages into 10 bins. The `table()` function then shows the frequency of players in each of these bins.
- The `cut()` function is also used in conjunction with **lattice graphics** to bin a date variable into months in the `bwplot()` function.

2. Shingles:

- A **shingle** is described as a generalization of a factor to a continuous variable. It consists of a numeric vector and a **set of intervals that are allowed to overlap**, similar to roof shingles.
- Shingles are **used extensively in the `lattice` package** as a way to easily use a continuous variable as a **conditioning or grouping variable**.

The `shingle()` function is used to create shingles in R:

```
shingle(x, intervals = sort(unique(x)))
```

- The `intervals` argument can be a numeric vector indicating the breaks or a two-column matrix where each row represents an interval.
- The `equal.count()` function can be used to create shingles where each bin contains the **same number of observations**.

3. Summarizing Functions (`tapply()` and `aggregate()` and `table()`/`xtabs()` in relation to binning):

- While `tapply()` and `aggregate()` are primarily for **summarizing data within groups**, they can be used in conjunction with binning. You would first use `cut()` to create a factor representing the bins, and then use `tapply()` or `aggregate()` to calculate summary statistics for each bin.
- The `table()` and `xtabs()` functions are used to create **contingency tables (frequency counts)** based on factor levels. After using `cut()` to bin a numeric variable into a factor, you can use these functions to see the distribution of the data across the created bins.

In summary, R provides powerful and flexible tools for binning data. The `cut()` function is a fundamental tool for creating bins from continuous variables, while shingles offer a more specialized approach often used with the `lattice` package for conditioning. Additionally, summarizing functions can be effectively used after binning to analyze the data within the defined intervals.

Subsets and Summarizing functions [Note : Related with Transformations / binning]

Both **subsets** and **summarizing functions** are important aspects of preparing data for analysis in R, as highlighted in your syllabus and detailed in Chapter 12 of "R in a Nutshell".

Subsets

Often, when working with data, you may need to focus on a specific portion of it rather than the entire dataset. This is where subsetting comes in handy. You might have too much data overall, or you might want to analyze specific groups of observations based on certain criteria, such as patient records for a particular age range and condition. Subsetting allows you to extract the relevant parts of your data for further analysis.

There are several ways to create subsets in R, according to "R in a Nutshell":

- **Bracket Notation**: You can subset data frames using **square brackets** `[]`. For data frames, you can select rows by providing a vector of logical values within the brackets. If a logical expression can describe the rows you want to select, you can use that expression as an index. For example, if you have a data frame named `batting.w.names` and want to select rows where `yearID` is equal to 2008, you could potentially use a logical condition within the brackets (though the source doesn't provide the exact bracket notation example, it sets up a `subset()` example based on this condition).

- **The `subset()` Function**: A more readable alternative to bracket notation is the **`subset()` function**. While anything achievable with `subset()` can also be done with bracket notation, `subset()` can make your code clearer by allowing you to directly use variable names from the data frame in your selection criteria. The `subset()` function has the following arguments:

    - `x`: The **object** (like a data frame, vector, or matrix) from which you want to calculate a subset.

- `subset`: A **logical expression** that describes which rows you want to return. Only rows where this expression evaluates to `TRUE` will be included in the subset.
- `select`: An **expression** indicating which columns you want to return. You can specify column names (as character strings or unquoted names) or use functions to help select columns.
- `drop`: This argument is passed to the `[` operator and its default is `FALSE`.

Examples of using `subset()` are provided:

```
 > library(nutshell) # Assuming batting.w.names is in this package

> data(Batting)

> batting.w.names <- Batting # For the example to run


> batting.w.names.2008 <- subset(batting.w.names, yearID==2008)

> head(batting.w.names.2008)

  playerID yearID stint teamID lgID   G  AB  R  H 2B 3B HR RBI SB CS BB SO IBB HBP SH SF GIDP

1  abercda01   2008   1    BOS  AL 127  11   1   2  0  0  0   0  0  0  0  4   3   0  0  0    0

2 aaronha01   2008   1    ATL  NL   9   0   0   0  0  0  0   0  0  0  0  0   0   0  0  0    0

3 aardsda01   2008   1    BOS  AL  47   0   0   0  0  0  0   0  0  0  0  0   0   0  0  0    0

4 aaronha01   2008   1    MIL  NL  10   0   0   0  0  0  0   0  0  0  0  0   0   0  0  0    0

5 abadan01   2008   1    HOU  NL  21   0   0   0  0  0  0   0  0  0  0  0   0   0  0  0    0

6 abeyta01   2008   1    SFN  NL  10   0   0   0  0  0  0   0  0  0  0  0   0   0  0  0    0


> batting.w.names.2008.short <- subset(batting.w.names, yearID==2008,

+     c("playerID","yearID","stint","teamID","G","AB","H"))

> head(batting.w.names.2008.short)

  playerID yearID stint teamID   G  AB  H

1  abercda01   2008   1    BOS 127  11   2

2 aaronha01   2008   1    ATL   9   0   0
```

```
3 aardsda01   2008   1    BOS  47   0   0

4 aaronha01   2008   1    MIL  10   0   0

5 abadan01   2008   1    HOU  21   0   0

6 abeyta01   2008   1    SFN  10   0   0
```

- The first example selects all columns for rows where `yearID` is 2008. The second example selects only the specified columns (`"playerID"`, `"yearID"`, `"stint"`, `"teamID"`, `"G"`, `"AB"`, `"H"`) for the same subset of rows.

Random Sampling: Sometimes, you might want to take a **random sample** of your data. This can be useful if you have a very large dataset and want to work with a more manageable subset for performance reasons or statistical validity. The `sample()` function can be used to take a random sample of observations.

```
 > # Assuming 'nrow(batting.w.names)' gives the total number of rows

> n_rows <- nrow(batting.w.names)

> sample_indices <- sample(1:n_rows, size = 100) # Take a random sample of 100 row indices

> batting.sample <- batting.w.names[sample_indices, ] # Subset the data frame using these indices

> head(batting.sample)

      playerID yearID stint teamID lgID  G AB R H 2B 3B HR RBI SB CS BB SO IBB HBP SH SF GIDP

27100  gathrje01   1999   1    FLO  NL 106 295 34 77 19 2  4  25  2  2 12 58   0   3  1  3    4

7085   bondsbo01   1990   1    PIT  NL 151 519 95 166 32 3 33 114  8  5 78 93   5   5  0  5    5

17687  joycema01   1994   1    KCA  AL  16  20 10  6  1 0  0   3  0  0  2  7   0   0  0  0    0

745    aldomdo01   1977   1    NYA  AL  21   1  0  0  0 0  0   0  0  0  0  1   0   0  0  0    0

9185   brackbi01   1986   1    ATL  NL  50   0  0  0  0 0  0   0  0  0  0  0   0   0  0  0    0

18355  lankfje01   1995   1    STL  NL 153 588 90 170 44 3 27  85  7  3 73 99   7   6  0  5   19
```

- The `sample()` function here generates a random selection of row numbers, which are then used to subset the data frame.

  For more complex random sampling methods like stratified sampling, the `sampling` package is mentioned.

Summarizing Functions

Summarizing functions in R allow you to aggregate and condense your data into a more concise and understandable form. This is often necessary when you have very detailed data, but you need to see overall trends or key statistics. "R in a Nutshell" describes several functions for this purpose:

`tapply()`: The `tapply()` function is a flexible way to apply a function to subsets of a vector, where the subsets are defined by the levels of one or more factor variables. Its syntax is:

```
tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

-
    - `X`: The **vector** you want to summarize.
    - `INDEX`: A **list of factors** (or objects that can be coerced to factors) of the same length as `X`. Each unique combination of factor levels in `INDEX` defines a subset of `X`.
    - `FUN`: The **function** to be applied to each subset of `X`. If `NULL`, `tapply()` returns a table of the subsets.
    - `...`: Optional arguments to be passed to `FUN`.
    - `simplify`: A logical value indicating whether the result should be simplified to an array if possible.

For example, to find the mean number of hits (`H`) for each team (`teamID`) in the `batting.2008` data (assuming you have this data frame):

```
> library(nutshell)

> data(batting.2008)

> mean_hits_by_team <- tapply(batting.2008$H, batting.2008$teamID, mean)

> head(mean_hits_by_team)

         ARI          ATL          BAL          BOS          CHA

2.716049e+02 2.700357e+02 2.675518e+02 2.794310e+02 2.625000e+02

         CLE

2.610989e+02
```

- This applies the `mean()` function to the `H` column, grouped by the unique values in the `teamID` column.

`aggregate()`: The `aggregate()` function is another powerful tool for summarizing data, especially within data frames. It can also be applied to time series. The syntax for data frames is:

```
aggregate(x, by, FUN, ...)
```

-
    - `x`: The **data frame** (or other data object) you want to aggregate.
    - `by`: A **list of grouping elements**, each as long as the rows of `x`. These typically are columns (or combinations of columns) that define the groups for aggregation.
    - `FUN`: A **scalar function** to be used to compute the summary statistic for each group.
    - `...`: Further arguments passed to `FUN`.

The example in "R in a Nutshell" shows how to use `aggregate()` to sum various batting statistics (`AB`, `H`, `BB`, `2B`, `3B`, `HR`) for each team (`teamID`):

```
> aggregate(x=batting.2008[, c("AB", "H", "BB", "2B", "3B", "HR")],
+   by=list(batting.2008$teamID), FUN=sum)
  Group.1   AB    H  BB  2B 3B  HR
1     ARI 5409 1355 587 318 47 159
2     ATL 5604 1514 618 316 33 130
3     BAL 5559 1486 533 322 30 172
4     BOS 5596 1565 646 353 33 173
5     CHA 5553 1458 540 296 13 235
...
```

- Here, the `by` argument is a list containing `batting.2008$teamID`, which specifies that the aggregation should be done for each unique team ID. The `FUN=sum` argument indicates that the `sum()` function should be applied to the specified columns for each team.

- **Counting Values**: You can also summarize data by counting the occurrences of different values or combinations of values using functions like **`table()`** and **`xtabs()`**. The `table()` function uses cross-classifying factors to build a contingency table of the counts at each combination of factor levels. For example, after binning a continuous variable using `cut()`, you can use `table()` to see the frequency of observations in each bin .

`summary()`: The **`summary()`** function is a generic function that provides a quick and useful overview of the data in a data frame or other R object. The output of `summary()` depends on the data type of each column. For numeric variables, it typically shows the minimum, first quartile, median, mean, third quartile, and maximum values. For factor variables, it shows the count of each level. For character variables, it usually provides the class and length.

```
 > summary(batting.2008$AB)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.

    0.0   101.0   333.0   303.7   499.0   701.0

> summary(as.factor(batting.2008$teamID))

   ARI  ATL  BAL  BOS  CHA  CLE  COL  DET  FLO  HOU  KCA  LAA  LAN  MIA

    33   34   33   35   33   33   35   34   32   33   32   34   33   31

   MIL  MIN  NYA  NYN  OAK  PHI  PIT  SDN  SEA  SFN  SLN  TBA  TEX  TOR

    34   33   33   34   33   34   33   34   34   32   34   33   35   33

   WAS

    33
```

- **`str()`**: While not strictly a summarizing function in the sense of aggregation, the **`str()`** function (for "structure") provides a concise overview of the structure of an R object, including its data type, dimensions, and the first few values. This can be helpful in understanding the overall makeup of your data.

These subsetting and summarizing techniques are fundamental for exploring and preparing your data for more advanced analyses in R.

Data Cleaning, Finding and removing Duplicates, Sorting.

Data Cleaning

Data cleaning is a crucial step in preparing data for analysis. It involves **identifying problems caused by data collection, processing, and storage processes and modifying the data so that these problems don't interfere with analysis**. Data cleaning **doesn't mean changing the meaning of data**.

"R in a Nutshell" emphasizes that even data from seemingly perfect sources like web servers or financial databases can have issues. It estimates that a significant portion (around 80%) of the effort in a typical data analysis project is spent on finding, cleaning, and preparing data.

The syllabus mentions "Data Cleaning" as a key step in Module 2 ("Reading and writing data"). It also lists "Missing Data - NA, NULL" as a related concept, as handling missing values is often a part of data cleaning. Course Outcome 2 includes a question to "Explain Data cleaning in R".

"R in a Nutshell" provides an example related to credit scores, where valid scores should be between 340 and 840, but the data contained values like 997, 998, and 999, which had special meanings like "insufficient data". Similarly, hospital patient data might have issues with the same doctor seeing multiple patients with the same name, leading to incorrect single records, or the same patient seeing multiple doctors, creating multiple records.

Therefore, data cleaning involves identifying and addressing such inconsistencies and errors to ensure the data is suitable for meaningful analysis.

Finding and Removing Duplicates

Data sources often contain duplicate values. Depending on how the data is used, duplicates can cause problems, so **checking for duplicates is a good practice if they aren't supposed to be there**. The syllabus lists "Finding and removing Duplicates" as a specific data preparation task.

"R in a Nutshell" highlights that if stock tickers are fetched multiple times accidentally, it will result in duplicate rows in the data. R provides useful functions for detecting duplicate values, such as the **`duplicated()` function**. This function **returns a logical vector showing which elements are duplicates of values with lower indices**. Applying `duplicated()` to a data frame will indicate which rows are duplicates of earlier rows.

You can use the logical vector returned by `duplicated()` to **remove duplicates using subsetting**:

```
> my.quotes.unique <- my.quotes.2[!duplicated(my.quotes.2),]
```

Alternatively, you can use the **`unique()` function** to directly **return a vector, data frame, or array with duplicate elements/rows removed**:

```
> my.quotes.unique <- unique(my.quotes.2)
```

Sorting

Sorting is another useful operation for analysis.

"R in a Nutshell" mentions sorting as one of the final operations that might be useful for analysis, along with ranking functions. The primary function for sorting in R is **`sort()`**. The `sort()` function **sorts (or orders) a vector or factor (partially) into ascending (or descending) order**.

For example, to sort a numeric vector `x` in ascending order:

```
> sort(x)
```

To sort in descending order, you can use the `decreasing = TRUE` argument:

```
> sort(x, decreasing = TRUE)
```

The `order()` function is also relevant for sorting. However, instead of returning the sorted vector itself, **`order()` returns a permutation that rearranges its first argument into ascending or descending order, breaking ties by further arguments**. This is particularly useful for sorting data frames, as you can use the indices returned by `order()` to rearrange the rows of the data frame based on the values in one or more columns.

For instance, to sort a data frame `df` based on the values in the column `age` in ascending order:

```
> df_sorted <- df[order(df$age), ]
```

To sort by `age` in descending order:

```
> df_sorted <- df[order(df$age, decreasing = TRUE), ]
```

You can also sort by multiple columns. For example, to sort by `age` (ascending) and then by `salary` (descending) within each age group:

```
> df_sorted <- df[order(df$age, -df$salary), ]
```

The `-` sign before `df$salary` indicates descending order for that column.

In summary, data cleaning involves addressing various data quality issues. Finding and removing duplicates can be done using `duplicated()` and `unique()`. Sorting can be achieved using `sort()` for vectors and factors, and `order()` for rearranging rows in data frames based on column values. These are fundamental steps in preparing data for effective analysis in R.