1. Overview

R packages are libraries that extend the functionality of R. They include functions, data, and documentation in a convenient format that is easy to install, reuse, and share with others.

Why Create R Packages?

- **Reusable code:** Complex analyses or algorithms can be encapsulated in functions, organized into a package, and reused in the future.
- **Process standardization:** Packages provide a standardized tool for data analysis that can be used across teams or projects.
- **Clear documentation:** Packages include usage examples and function descriptions, making them easier to understand.
- **Integration:** Packages easily integrate with each other, creating an ecosystem of tools.
- **Public availability:** Packages can be shared with colleagues or published in public repositories like CRAN, GitHub, or Bioconductor.
- **Ease of installation:** Users can quickly install a package using commands like:

```
install.packages("")
devtools::install_github("")
BiocManager::install("") # from Bioconductor
```

Popular R Packages for Bioinformatics (Bioconductor Repository)

- ***edgeR, DESeq2, Limma:*** *For differential gene expression analysis.*
- ***GenomicRanges:*** *For working with genomic coordinates.*
- ***Biostrings:*** *For analyzing DNA, RNA, and protein sequences.*
- ***tidyverse:*** *Although not specific to bioinformatics, it is widely used for data manipulatio bioinformatics (e.g., dplyr, ggplot2).*
- ***phyloseq:*** *For microbiome data analysis.*
- ***pheatmap:*** *For creating heatmaps, commonly used in bioinformatics.*

R Packages for Creating R Packages :)

- **devtools:** Simplifies the creation, development, and testing of packages.
  Functions: `create()`, `install()`, `check()`.
- **roxygen2:** For generating documentation.
  Documentation is written as comments in the code, and a `NAMESPACE` file is automatically generated.
- **usethis:** Automates package structure creation.
  Creates folders (`R/`, `tests/`, `man/`), `README` templates, `LICENSE` files, and more.
- **testthat:** For writing tests.
  Helps with testing package functions.
- **pkgdown:** For generating a package website with documentation.
- **rcmdcheck:** Checks the package for compliance with CRAN standards.

2. Building Structure of an R Package

Package	Function	Description
usethis	`usethis::create_package(path)`	*Creates the basic structure of an R package.*
	`usethis::use_readme_rmd()`	*Adds a README file in RMarkdown format.*
	`usethis::use_testthat()`	*Sets up a folder for unit testing.*
	`usethis::use_package(package)`	*Adds the specified package as a dependency in the DESCRIPTION file.*
	`usethis::use_git()`	*Initializes a Git repository for the package.*
devtools	`devtools::document()`	*Generates documentation (.Rd files and NAMESPACE).*
	`devtools::check()`	*Checks the package for errors and warnings.*
	`devtools::install()`	*Installs the current version of the package locally.*
	`devtools::load_all()`	*Loads all functions from the package without installation (for debugging).*
	`devtools::build()`	*Creates a package archive (.tar.gz) for publication.*
roxygen2	`roxygen2::roxygenise()`	*Generates documentation and a NAMESPACE file based on #' comments.*
	`roxygen2::update_collate()`	*Updates the file order in the DESCRIPTION if necessary.*
	`roxygen2::roxygen2_options()`	*Configures settings for automatic documentation generation.*
testthat	`testthat::test_dir(path)`	*Runs all tests in the specified directory.*
	`testthat::expect_equal(object, expected)`	*Checks if the object equals the expected value.*
	`testthat::test_that(desc, code)`	*Defines a set of tests with a description and test scenarios.*
desc	`desc::desc_set(field, value)`	*Sets the value for a specified field in the DESCRIPTION file.*
	`desc::desc_add_dep(package, type)`	*Adds a dependency to Imports, Suggests, or another field.*
	`desc::desc_get(field)`	*Retrieves the value of a specified field from the DESCRIPTION file.*
rcmdcheck	`rcmdcheck::rcmdcheck()`	*Checks the package as CRAN does.*
	`rcmdcheck::check_details()`	*Extracts details of the check (errors, warnings, notes).*
pkgdown	`pkgdown::build_site()`	*Creates a website for the package documentation.*
	`pkgdown::build_reference()`	*Builds the documentation section for all functions.*
	`pkgdown::build_news()`	*Generates the package news page (Changelog).*

1. Create the Package Directory

```
install.packages("usethis")
usethis::create_package("path/to/new_package")
```

This command creates a new folder named new_package with essential files and subdirectories.

Files and Directories Created

After running the command, the following structure generated:

```
new_package/
├── DESCRIPTION       # Metadata about your package
├── NAMESPACE         # Declares exported/imported functions
├── R/                # Contains R scripts (your functions)
├── man/              # Documentation (generated automatically)
├── tests/            # Test files for your functions
```


2.Edit the DESCRIPTION File

The DESCRIPTION file is the heart of your package metadata. It contains:

- Package name: Must be unique.
- Version: Use semantic versioning (e.g., 0.1.0).
- Author/Maintainer: Include name and email.
- License: Specify the license (e.g., GPL-3, MIT).

Example DESCRIPTION file:

```
Package: new_package
Type: Package
Title: Example R Package
Version: 0.1.0
Author: Name <email@example.com>
Maintainer: Name <email@example.com>
Description: A short description of what the package does.
License: MIT
```

```
Encoding: UTF-8
LazyData: true
```

3. Set Up the R/ Directory

The `R/` folder is where all my R scripts (functions) are stored. Use `roxygen2` for function documentation. Example:

```
#' My Example Function
#'
#' This function adds two numbers together.
#' @param x First number.
#' @param y Second number.
#' @return Sum of `x` and `y`.
#' @export
my_function <- function(x, y) {
    return(x + y)
}
```

4.Edit the NAMESPACE File

Use `roxygen2` to generate the NAMESPACE file automatically:

```
devtools::document()
```

Add Dependencies

If my package depends on other packages, declare them in the `Imports` field of the DESCRIPTION file:

```
Imports:
    ggplot2,
    dplyr
```

Then, use `usethis::use_package()` to add dependencies:

```
usethis::use_package("ggplot2")
```

5.Add Unit Tests

Create a `tests/` folder using `usethis::use_testthat()`:

```
usethis::use_testthat()
```

Write test cases in the `tests/testthat/` folder, e.g., `test_my_function.R`:

```
test_that("my_function works correctly", {
    expect_equal(my_function(2), 4)
})
```

3. Using roxygen2 for R Package Documentation

The `roxygen2` library allows automating the process of generating documentation. Instead of manually writing `.Rd` files, `roxygen2` uses special comments directly in the code. These comments are then turned into documentation files in the `man/` directory.

Structure of the R/ Directory

The `R/` directory contains the functions of your package. Each function is stored in a separate `.R` file. Documentation is written directly in these `.R` files, above the function definition.

Special comments start with `#'` and include metadata about the function.

Example of a Function with Documentation

```
#' Summation of Two Numbers
#'
#' This function takes two numbers as input and returns their sum.
#'
#' @param a A numeric value. The first number.
#' @param b A numeric value. The second number.
#' @return The sum of `a` and `b`.
#' @examples
#' add_numbers(2, 3) # Returns 5
#' @export
add_numbers <- function(a, b) {
  return(a + b)
}
```

Explanation of Tags

- `@param` : Describes the function's parameters.
- `@return` : Describes what the function returns.
- `@examples` : Includes examples of how to use the function.
- `@export` : Indicates that the function will be available to users of the package.

Generating Documentation

After writing the documentation, generate the `.Rd` files:

```
devtools::document()
```

This will create a documentation file in the `man/` folder of your package. For example, for the `add_numbers` function, a file `man/add_numbers.Rd` will be created.

Checking Documentation

Ensure the documentation is displayed correctly:

1. Load the package:

   ```
   devtools::load_all()
   ```

2. Check the documentation in the console:

   ```
   ?add_numbers
   ```

All `roxygen2` Tags

- *`@title` : Short title of the function.*
- *`@description` : Detailed description.*
- *`@details` : Additional information.*
- *`@seealso` : References to related functions or documentation.*
- *`@import` : Specifies which functions to import from other packages.*
- *`@export` : Makes the function available to users of the package.*
- *`@importFrom` : Import a specific function from a package.*
- *`@note` : Notes or warnings for the user.*
- *`@author` : Author of the function.*
- *`@format` : Data format.*
- *`@source` : Source of the data.*
- *`@references` : References or citations.*
- *`@examplesIf` : Examples executed under specific conditions.*
- *`@testexamples` : Test the examples provided in the documentation.*
- *`@family` : Group related functions together.*
- *`@name` : Name of the object or topic.*
- *`@docType` : Documentation type (e.g., package, function, data).*
- *`@concept` : Concept tag for classification.*
- *`@aliases` : Alternative names for the function.*
- *`@usage` : Custom usage instructions.*
- *`@section` : Custom sections for documentation.*
- *`@keywords` : Keywords for indexing.*
- *`@inherit` : Inherit documentation from another function.*
- *`@inheritParams` : Inherit parameter descriptions from another function.*

- *`@rawNamespace` : Raw namespace instructions.*
- *`@examplesDontRun` : Examples that should not be executed.*
- *`@include` : Include code or documentation from another file.*

4. Incorporating Data into an R Package

Adding data to an R package allows distributing it along with functions. This is useful for including examples, tests, or predefined datasets.

1. Preparing the Data

Before adding data to a package:

- Data must be in a format supported by R (e.g., `.csv`, `.rds`, or R objects like `data.frame`).
- Data should be clean and documented.

2. Saving Data to the `data/` Directory

Data included in a package is saved in the `data/` folder as `.rda` files.

Example of adding data:

```
my_data <- data.frame(
    id = 1:5,
    value = c(10, 20, 30, 40, 50)
)
# Save as an .rda file
save(my_data, file = "data/my_data.rda")
```

Place the `my_data.rda` file in the `data/` folder of your package.

3. Documenting the Data with roxygen2

Data documentation is mandatory to ensure users can utilize it correctly.

Example of data documentation:

Add a description in an `.R` file in the `R/` folder (e.g., `my_data.R`):

```
#' Example Dataset
#'
#' This dataset contains sample data for demonstration purposes.
#'
#' @format A data frame with 5 rows and 2 variables:
#' \describe{
#'   \item{id}{Numeric ID of the sample.}
#'   \item{value}{Numeric value associated with the sample.}
#' }
#' @source Generated manually for package examples.
"my_data"
```

Explanation of Tags

- `@format`: Describes the structure of the data.
- `@source`: Specifies the data source.
- `@describe`: Provides detailed descriptions of columns (variables).

Then, generate (or update) the documentation:

```
devtools::document()
```

4. Verifying the Data

Ensure the data is accessible:

```
data("my_data")
?my_data
```

Dynamic Data

If the data needs to be generated by the user at runtime, use the `inst/` folder:

- Place the source data files in `inst/extdata/`.
- These files can be loaded using:

```
file_path <- system.file("extdata", "my_file.csv", package = "MyPackage")
data <- read.csv(file_path)
```

Example with Different Types of Data

Adding a table (`data.frame`):

```
sample_data <- data.frame(
    gene = c("Gene1", "Gene2", "Gene3"),
    expression = c(1.2, 3.4, 2.8)
)
save(sample_data, file = "data/sample_data.rda")
```

Adding RDS files:

Save an object in `.rds` format:

```
saveRDS(sample_data, file = "inst/extdata/sample_data.rds")
```

Tips

- **Data Size:** Data should be small (CRAN limits package size to 5 MB). For larger data, use `inst/extdata/`.
- **Documentation:** Always describe the format and source of the data.
- **Testing:** Ensure the data loads and displays correctly.

5. Testing the R Package

Testing R packages is an essential part of development, ensuring that functions work correctly and as expected. The `testthat` library is widely used in R for writing and running tests, providing a convenient interface for package testing.

Why Test a Package?

- **Ensure function correctness:** Verify that functions return expected results.
- **Simplify refactoring:** Tests alert you to unintended errors during modifications.
- **Increase reliability:** Confirm that functions handle all intended cases (and do not fail on unexpected inputs).

Steps for Testing a Package

1. Installing `testthat`

Ensure that the `testthat` library is installed:

```
install.packages("testthat")
```

2. Setting Up Testing in the Package

Use a function from `usethis` to initialize testing:

```
usethis::use_testthat()
```

This command will create a `tests/` folder in your package and set up the basic structure for integrating with the test system.

3. Writing Tests

Each function in your package should have a corresponding test file in the `tests/testthat/` folder.

Example:

If you have a function `add_numbers`, create a file `tests/testthat/test-add_numbers.R`:

```
test_that("add_numbers works correctly", {
  # Check correct results
  expect_equal(add_numbers(2, 3), 5)
  expect_equal(add_numbers(-1, 1), 0)

  # Check for errors
  expect_error(add_numbers("a", 1)) # Cannot add text and numbers
})
```

Explanation:

- `test_that`: The main test block describing what is being tested.
- `expect_*`: A family of functions to verify results:
  - `expect_equal`: Checks that the result matches the expected value.
  - `expect_error`: Checks that a function throws an error.
  - `expect_true / expect_false`: Checks that the result is `TRUE` or `FALSE`.

4. Running Tests

Run all tests in the package:

```
devtools::test()
```

Advanced Testing Features

Testing Edge Cases

Ensure that functions handle boundary values correctly:

```
test_that("add_numbers handles edge cases", {
  expect_equal(add_numbers(0, 0), 0)
  expect_equal(add_numbers(Inf, -Inf), 0)
})
```

Testing Data

If your function works with data, verify that the data structure remains valid:

```
test_that("function returns valid data structure", {
  result <- my_function(data_frame)
  expect_true(is.data.frame(result))
  expect_named(result, c("col1", "col2", "col3"))
})
```

Testing Performance

Check that a function runs within a reasonable time:

```
test_that("function runs quickly", {
  expect_silent(system.time(my_function(data_frame)) < 1)
})
```

Test Report

When you run `devtools::test()`, `testthat` generates a detailed report:

- Successful tests.
- Failed tests with detailed information about the causes.

Organizing Tests

- Create a separate test file for each function: `test-function_name.R`.
- Divide tests into logical blocks using `test_that`.

Example: Full Test Structure

Function

```
#' Add Two Numbers
#' @param a First number
#' @param b Second number
#' @return Sum of a and b
#' @export
add_numbers <- function(a, b) {
  if (!is.numeric(a) || !is.numeric(b)) {
    stop("Inputs must be numeric")
  }
  a + b
}
```

Tests (File: `tests/testthat/test-add_numbers.R`)

```
test_that("add_numbers works correctly", {
  expect_equal(add_numbers(2, 3), 5)
  expect_equal(add_numbers(-1, 1), 0)
  expect_error(add_numbers("a", 1))
})

test_that("add_numbers handles edge cases", {
  expect_equal(add_numbers(0, 0), 0)
  expect_equal(add_numbers(Inf, -Inf), 0)
})
```

Tips

- Cover edge cases thoroughly.
- Use meaningful names for tests.
- Add tests for every function update or change.
- Test both exported and internal functions. For internal functions, use:

```
pkgload::load_all(export_all = FALSE)
```

Adding Advanced Features to an R Package

Advanced features in R packages enhance functionality and usability. Below are examples of such features:

1. Interactive Shiny Applications

Embed Shiny applications directly into my package by creating a `shiny/` folder.

More about using Shiny with R can be found here: Shiny Basics.

Steps to Embed a Shiny App in My R Package

Prepare Package

Ensure my package structure is set up using tools like `usethis`:

```
usethis::create_package("MyShinyPackage")
```

Add the Shiny App Files

Create a `shiny/` directory and add the following files:

- **ui.R:** Contains the user interface definition.
- **server.R:** Contains the server logic.

Example of `shiny/ui.R`:

```
shiny::fluidPage(
  shiny::titlePanel("Example Shiny App"),
  shiny::sidebarLayout(
    shiny::sidebarPanel(
      shiny::sliderInput("num", "Choose a number:", 1, 100, 50)
    ),
    shiny::mainPanel(
      shiny::plotOutput("plot")
    )
  )
)
```

Example of `shiny/server.R`:

```
function(input, output, session) {
  output$plot <- shiny::renderPlot({
    hist(rnorm(input$num))
  })
}
```

Add a Function to Run the App

Create a wrapper function in my `R/` directory to run the Shiny app:

```
#' Run the Example Shiny App
#'
#' This function launches the Shiny app embedded in the package.
#' @export
run_shiny_app <- function() {
  app_dir <- system.file("shiny", package = "MyShinyPackage")
  if (app_dir == "") {
    stop("Could not find Shiny app directory. Please reinstall the package.", call. = FALSE)
  }
  shiny::runApp(app_dir, display.mode = "normal")
}
```

Add Dependencies

Declare Shiny as a dependency in the `DESCRIPTION` file:

```
Imports:
    shiny
```

Test the App Locally

Build and load my package, then run the app using my wrapper function:

```
devtools::load_all()
run_shiny_app()
```

Additional Features for Shiny Apps

- **Embed Data:** Store datasets in `inst/extdata/` and access them dynamically:

```
data_path <- system.file("extdata", "example_data.csv", package = "MyShinyPackage")
data <- read.csv(data_path)
```

- **Customize Appearance:** Add custom CSS and JavaScript:

```
shiny::tags$head(
  shiny::includeCSS("www/styles.css")
)
```

- **Interactive Help with Vignettes:** Link my Shiny app to a vignette:

```
usethis::use_vignette("shiny_app_guide")
```

Testing My App

Use the `shinytest` package to automate testing:

```
install.packages("shinytest")
shinytest::recordTest("path/to/shiny/app")
```

Example Structure of a Package with Shiny App

```
MyShinyPackage/
├── R/
│   └── run_shiny_app.R
├── DESCRIPTION
├── NAMESPACE
├── inst/
│   └── shiny/
│       ├── ui.R
│       ├── server.R
│       └── www/ (optional, for assets like CSS/JS)
└── vignettes/
    └── shiny_app_guide.Rmd
```

2. S3 and S4 Classes

S3 Classes

Create flexible object-oriented behavior using the `class` attribute:

```
my_object <- list(data = 1:10)
class(my_object) <- "my_class"
print.my_class <- function(x) {
  cat("This is a custom class object\n")
  print(x$data)
}
```

S4 Classes

Use formal class definitions for stricter validation:

```
setClass("Person",
  slots = list(name = "character", age = "numeric")
)
```

3. Custom Operators

Define new operators with special symbols:

```
"%add%" <- function(a, b) a + b
5 %add% 10  # Returns 15
```

4. Integrating C++ Code

For performance, integrate C++ with Rcpp:

```
install.packages("Rcpp")
usethis::use_rcpp()
```

Write C++ code in `src/` :

```
// [[Rcpp::export]]
double add_numbers(double a, double b) {
    return a + b;
}
```

Use in R:

```
Rcpp::sourceCpp("src/add_numbers.cpp")
```

5. Adding Vignettes

Vignettes provide long-form package documentation:

```
 usethis::use_vignette("introduction")
devtools::build_vignettes()
```

Sharing an R Package

Sharing an R package ensures that others can access, use, and contribute to it. Below are common methods for distributing R packages: GitHub, CRAN, and local sharing.

Methods for Sharing

1. Sharing via GitHub

GitHub is a popular platform for hosting and distributing R packages, especially during development or for pre-release versions.

Steps:

- **Push the package to a GitHub repository:**

```
# Initialize a Git repository
git init
git add .
git commit -m "Initial commit"

# Create a repository on GitHub and push the code
git remote add origin https://github.com/username/packagename.git
git branch -M main
git push -u origin main
```

- **Enable installation via GitHub:**

```
install.packages("remotes")
remotes::install_github("username/packagename")
```

- **Add a README file:** Include installation instructions and a summary of functionality:

```
usethis::use_readme_md()
```

2. Publishing on CRAN

CRAN (Comprehensive R Archive Network) is the primary repository for R packages, ensuring accessibility to all R users.

Steps:

- **Prepare the package:** Confirm compliance with CRAN requirements:

```
devtools::check()
```

Address any errors, warnings, or notes.

- **Submit to CRAN:**

```
devtools::release()
```

Follow the instructions to upload the package through the CRAN submission portal.

- **Maintain updates:** Ensure compatibility with new R versions and respond to CRAN feedback if required.

3. Sharing Locally

Local sharing is useful for small teams or offline use.

Steps:

- **Build the package:**

```
devtools::build()
```

This generates a `.tar.gz` file (e.g., `mypackage_1.0.0.tar.gz`).

- **Distribute the file:** Provide the file via email, shared drive, or another platform.
- **Install locally:**

```
install.packages("path/to/mypackage_1.0.0.tar.gz", repos = NULL, type = "source")
```

4. Publishing on Bioconductor

For bioinformatics-focused packages, Bioconductor is often preferred.

Steps:

- **Prepare for submission:** Ensure compliance with Bioconductor guidelines, including proper dependency usage.
- **Submit the package:** Create an issue in the Bioconductor submission tracker.

Best Practices for Sharing

Versioning:

- Update the `DESCRIPTION` file with semantic versioning (1.0.0, 1.0.1, etc.).
- Increment versions for every release.

Documentation:

- Include a README file with installation instructions and usage examples:

```
usethis::use_readme_md()
```

- Add vignettes for detailed examples:

```
usethis::use_vignette("example_vignette")
```

Licensing:

Specify a license using:

```
usethis::use_mit_license("Name")
```

Testing:

Ensure thorough test coverage using `testthat`:

```
devtools::test()
```

CI/CD Integration:

Automate testing and package checks with GitHub Actions:

```
usethis::use_github_action_check_standard()
```

Example Workflow

Sharing via GitHub:

```
usethis::use_github()
```

Publishing to CRAN:

```
devtools::check()
devtools::release()
```

Local Sharing:

```
devtools::build()
install.packages("path/to/package.tar.gz", repos = NULL, type = "source")
```

Steps to Create and Share an R Package

Step 1: Install Required Packages

```
install.packages(c("devtools", "usethis", "roxygen2", "testthat"))
```

Step 2: Create a Package Structure

```
usethis::create_package("path/to/MyPackage")
```

Step 3: Add Git (Optional, for Version Control)

```
usethis::use_git()  # Not required, can initialize manually via `git init`
```

Step 4: Add a License and README

```
usethis::use_mit_license("Your Name")
usethis::use_readme_md()
```

Step 5: Add a Function

```
usethis::use_r("add_numbers")  # Creates R/add_numbers.R
```

```
# Edit R/add_numbers.R to include:
#' Add Two Numbers
#' @param a First number
#' @param b Second number
#' @return Sum of `a` and `b`
#' @examples add_numbers(2, 3)
#' @export
add_numbers <- function(a, b) { a + b }
```

Step 6: Document the Package

```
devtools::document()
```

Step 7: Add Tests

```
usethis::use_testthat()  # Initializes tests/testthat/
```

```
# Create tests/testthat/test-add_numbers.R:
test_that("add_numbers works", {
  expect_equal(add_numbers(2, 3), 5)
  expect_error(add_numbers("a", 1))
})
```

Step 8: Run Tests

```
devtools::test()
```

Step 9: Add Data (Optional)

```
my_data <- data.frame(x = 1:5, y = 6:10)
save(my_data, file = "data/my_data.rda")
```

```
# Document the data in R/my_data.R:
#' Example Dataset
#' @format A data frame with 5 rows and 2 variables
#' @examples data(my_data)
"my_data"
```

Step 10: Check the Package

```
devtools::check()
```

Step 11: Build the Package

```
devtools::build()
```

Step 12: Share the Package

```
usethis::use_github()  # Optional, for GitHub sharing
devtools::release()    # For CRAN submission
```