

1 R Package

Check `pkgtest` repository on GitHub <https://github.com/ptds2024/pkgtest>.

Below we provide a detailed explanation of how to create and develop an R package. To begin, prepare the initial structure of the package.

Problem 1.

1. One should create a project: New project/New directory/R package (it is recommended to click "create a git repository" to later publish the package on github) or using a command `create_package("path/to/package_name")` on a console.
2. Create a remote repository on github and connect it to your package.
3. Install packages `devtools`, `usethis`, `knitr`, `pkgdown`, `roxygen2` and `testthat`, which are used a lot while developing a package.
4. To efficiently develop a package you should interact a lot with the console to write various commands.
5. There is no need to store default "hello.R", "hello.Rd" and "NAMESPACE" files, they can be removed (the new files appear during development of the package).

To create a function in the package:

1. Use a command `usethis::use_r("your_function_name")`, which makes an empty file "your_function_name.R" in the folder "R".
2. Add body of a function:

```
~%r%` <- function(y, x) {  
  fit <- lm(y ~ x)  
  coef(fit)  
}
```

3. Create a documentation block for the function. To this end, go here: Code/Insert Roxygen Skeleton. Please note that your cursor should be on the same string as the beginning of your function (not before, otherwise error might appear). The obtained file should like

```
#' Title  
#'  
#' @param y  
#' @param x  
#'  
#' @return  
#' @export  
#'  
#' @examples  
~%r%` <- function(y, x) {
```

```
fit <- lm(y ~ x)
coef(fit)
}
```

4. Edit necessary fields: replace "Title" with `@title` and write the appropriate title, fill in information about parameters, return result. `@export` means that the function will be available for users. Make examples with `@example` or `@examples` to illustrate for users how the function works (see Problem 6 for more details). Optionally one can add section `@description` to describe the function.
5. Since functions `lm` and `coef` belong to a built-in library `stats` we should mention that we use this functions in `@importFrom` (or `@import` to import whole packages). This section is not generated automatically.
6. The file should have the following view:

```
#' @title Function to calculate the regression coefficients
#' @description This function calculates
#' the regression coefficients of a linear model
#' @param y The dependent variable
#' @param x The independent variable
#' @return The regression coefficients
#' @example /inst/examples/eg_reg_coef.R
#' @importFrom stats lm coef
#' @export
`%r%` <- function(y, x) {
  fit <- lm(y ~ x)
  coef(fit)
}
```

7. Note that to create several functions in one package, one should create several separate files for each function (using the command `usethis::use_r("your_function_name")`).

Problem 2.

The `DESCRIPTION` file contains the metadata of a package (such as the author of the package, license, dependencies, etc.). It allows R to understand the package's dependencies and provides necessary metadata for users.

To choose a license use the following command: `usethis::use_mit_license`. The file should like

```
Package: pkgtest
Type: Package
Title: Package to showcase package building in R to students
Version: 0.1.0
Authors@R: c(person("Samuel", "Orso", email = "samuel.orso@unil.ch",
  role = c("aut", "cre")),
```

```
    person("Timofei", "Shashkov",
      email = "timofei.shashkov@unil.ch", role = "aut"))
Maintainer: <samuel.orso@unil.ch>
Description: More about what it does (maybe more than one line)
Use four spaces when indenting paragraphs within the Description.
License: MIT + file LICENSE
Suggests:
testthat (>= 3.0.0),
knitr,
rmarkdown
Depends: R (>= 4.0.0)
Encoding: UTF-8
LazyData: true
RoxygenNote: 7.3.1
Config/testthat/edition: 3
```

Problem 3.

To provide users with information about a package's functions and datasets, each package should include `.Rd` files, which are stored in the `"man"` folder.

To generate documentation for functions and datasets, you can use the command `devtools::document()`. This command automatically creates the necessary documentation files for the package and updates the `NAMESPACE` file, which manages which functions and objects are exported (made accessible to users) and which functions are imported from other packages.

To access the documentation for a created function `your_function`, use the command `?your_function`.

Problem 4.

To add a dataset, we first need to upload the raw dataset. This can be done using the following procedure:

1. Use the command `usethis::use_data_raw()` to create a folder called `data-raw` with an R script file named `DATASET.R`. Alternatively, you can create the folder and R script manually (although this is not recommended).
2. Upload the dataset `snipes.csv` (which you can find here <https://ptds.samorso.ch/exercises/>) to the `data-raw` folder.
3. Modify `DATASET.R`: Load the dataset using the command `read.csv` and then save it to the `data` folder as an `.rda` file, so it will be easily accessible for users after loading the package. Run the code in `DATASET.R` to save the dataset.
4. The resulting `DATASET.R` file should look like this:

```
## code to prepare snipes.csv dataset
snipes <- read.csv(file = "data-raw/snipes.csv")
usethis::use_data(snipes, overwrite = TRUE)
```

5. Add documentation for the dataset. To do this, create an R script file in the R folder with the following content:

```
#' Snipes price data
#'
#' @format ## snipes
#' A data frame with 48 rows and 3 columns:
#' \describe{
#'   \item{discount}{Discounted price of sneakers}
#'   \item{brand}{Brand of sneakers}
#'   \item{price}{Original price of sneakers}
#' }
#' @source <https://www.snipes.ch/>
"snipes"
```

6. To update the documentation, use the command `devtools::document()`.

Problem 5.

Another important component of each package is a *vignette*, which is an RMarkdown file used to provide a detailed guide on how to use the package. To create a vignette with the name "my-vignette", use the command `usethis::use_vignette("my-vignette")`. Modify the file to explain to users how to work with your package.

In order to run the rmarkdown file you should use the command `devtools::install()` to install the package on your computer.

Problem 6.

There are two different ways to provide examples in the documentation of functions. The first method is to include example calculations directly in the R script file for the functions. This is done using the `@examples` tag in the roxygen2 comments, as shown in the example below:

```
#' @title Function to calculate the regression coefficients
#' @description This function calculates the regression
# coefficients of a linear model
#' @param y The dependent variable
#' @param x The independent variable
#' @return The regression coefficients
#' @examples cars$speed~cars$distance
#' @importFrom stats lm coef
#' @export
`~` <- function(y, x) {
  fit <- lm(y ~ x)
  coef(fit)
}
```

Alternatively, for complex examples, you can create them as R scripts in the directory `inst/examples/`.

To start, create the nested folders and an R script either manually or by using the command `usethis::use_directory("inst/examples")`.

In the R script (e.g., `inst/examples/my_example.R`), you can write examples as before, which will be available for users to run. These examples demonstrate how to use your functions in different scenarios.

```
## linear regression
cars$speed %r% cars$dist
```

To document such examples, you should reference the file path `inst/examples/my_example.R` next to the `@example` tag (note: use `@example` for file-based examples, not `@examples` as used for inline examples).

```
#' @title Function to calculate the regression coefficients
#' @description This function calculates
#' the regression coefficients of a linear model
#' @param y The dependent variable
#' @param x The independent variable
#' @return The regression coefficients
#' @example /inst/examples/eg_reg_coef.R
#' @importFrom stats lm coef
#' @export
`%r%` <- function(y, x) {
  fit <- lm(y ~ x)
  coef(fit)
}
```

Before publishing a package, it is important to verify that it works correctly. First, we should *check* the package to ensure it meets R package standards and can be distributed without issues. This process covers a broad range of aspects, including documentation, dependencies, examples, and compliance with CRAN policies.

To ensure that functions work correctly, we should also add *tests*. These tests help confirm that the package functions as expected and can handle a variety of inputs and use cases.

Problem 7.

Before testing functions, we need to create test files, which will be located in `tests/testthat/`. By running the command `usethis::use_testthat()`, we create the directory `tests/testthat` along with a file `testthat.R` inside the `tests` folder. This file will manage the tests for the functions in the package.

Tests are written as R scripts located in the `testthat` folder. Common functions for testing include:

- `expect_error`: checks that an error is thrown for specific inputs.
- `expect_type`: verifies that the output type matches the expected type.
- `test_that`: organizes the tests for a function or feature.

Here is an example of a test file:

```
test_that("regression coefficient input check",{
  expect_error(cars$speed %r% cars)
})
test_that("regression coefficient output",{
  expect_type(cars$speed %r% cars$dist, "double")
})
```

To actually test the functions, run the command `devtools::test()`.

Problem 8.

To enable automated checking, use `usethis::use_github_action_check_standard()`. This command creates a `.github` folder that contains a `workflows` folder with an `R-CMD-check.yaml` file.

This YAML file configures GitHub Actions to automatically check the package on various operating systems and R versions each time updates are pushed to the remote repository. If any errors appear, GitHub will notify you.

Problem 9.

To create a professional website for your package, you can follow these steps:

1. Run the command `usethis::use_pkgdown()` to create the file `_pkgdown.yml`, which configures the website for your package.
2. Use `pkgdown::build_site()` to build the website locally.
3. To link the website with the remote GitHub repository, add the repository URL in `_pkgdown.yml` and include the same link in the `DESCRIPTION` file (e.g., URL: `<link>`). Don't forget to save these changes.
4. To set up automatic website updates via GitHub Actions, run the command `usethis::use_github_action("pkgdown")`.
5. Push the changes to your remote GitHub repository.

2 Object-Oriented Programming

Problem 1: Create a summary function for the class `pixel`

In R, method dispatch for S3 objects works by calling the method that matches the class of the object. First, we define a simple class `pixel`, then we create a custom `summary` method for this class.

```
# Define class 'pixel'
pixel <- function(x, y, color) {
  structure(list(x = x, y = y, color = color), class = "pixel")
}
```

```
# Define default summary method
summary.default <- function(object) {
  print("No summary available for this object.")
}

# Before implementing summary for 'pixel'
p <- pixel(10, 15, "red")
summary(p) # Will call the default summary

## [1] "No summary available for this object."

# Define summary method for 'pixel'
summary.pixel <- function(object) {
  cat("Pixel Information: \n")
  cat("X position: ", object$x, "\n")
  cat("Y position: ", object$y, "\n")
  cat("Color: ", object$color, "\n")
}

# After implementing summary for 'pixel'
summary(p) # Will call the pixel summary method

## Pixel Information:
## X position: 10
## Y position: 15
## Color: red
```

Problem 2: Difference between `t.test()` and `t.data.frame()`

The function `t.test()` performs a Student's t-test for statistical inference, while `t()` is a generic method that computes the transpose of a matrix or data frame. If you create an object with a custom class, and the class does not have a specific `t()` method, R will fall back to the default method.

Running the code below demonstrates the error due to the absence of a `t.test` method for the custom class:

```
x <- structure(1:10, class = "test")
t(x)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    2    3    4    5    6    7    8    9    10
## attr(,"class")
## [1] "test"
```

Since `t()` is looking for a method matching the class `test`, but no such method exists, it returns an error.

Problem 3: Understanding UseMethod()

In this example, `UseMethod()` is used to dispatch based on the class of the argument. The function `g()` does not modify the class of its arguments, so the default method `g.default()` is called. However, there is a subtle scoping issue in this code where the variable `y` is not available in the scope of `g.default()`.

```
g <- function(x) {  
  x <- 10  
  y <- 10  
  UseMethod("g")  
}  
  
g.default <- function(x) c(x = x, y = y)  
  
x <- 1  
y <- 1  
g(y) # Will throw an error because 'y' is not found in g.default's scope  
  
##   x   y  
##  1  10
```

To fix this, `y` must be passed as an argument or explicitly declared in the parent environment.

Problem 4: Understanding NextMethod()

In this example, the class of the object is changed within the function `generic2.b`, and `NextMethod()` is used to invoke the next method in the inheritance hierarchy.

```
generic2 <- function(x) UseMethod("generic2")  
  
generic2.a1 <- function(x) "a1"  
generic2.a2 <- function(x) "a2"  
  
generic2.b <- function(x) {  
  class(x) <- "a1"  
  NextMethod()  
}  
  
generic2(structure(list(), class = c("b", "a2")))  
  
## [1] "a2"
```

When `generic2.b()` is called, it modifies the class of `x` to `"a1"` and calls `NextMethod()`. Even though the class of `x` was modified, the function `NextMethod()` will dispatch to `generic2.a2()`, since the next element in the initial class list was `a2`. The final result will be `"a2"`.

3 Functional Programming

Problem 1: Using a for loop, calculate the total monthly sales for each product.

```
# 1. For loop approach

product_sales <- list(
  product1 = c(50, 45, 60, 55, 70, 80, 75, 90, 85, 60, 70, 65, 70, 75, 80,
               85, 90, 95, 85, 70, 75, 80, 60, 45, 55, 50, 45, 60, 65),
  product2 = c(30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100,
               105, 110, 115, 120, 125, 130, 135, 140, 145, 150, 155, 160,
               165, 170, 175),
  product3 = c(20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48,
               50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78)
)

# Initialize an empty vector to store results
f_for = function(x){total_sales <- c()
# Loop through each product in the product_sales list
for (product in names(x)) {
  # Calculate the total sales for the current product
  total_sales[product] <- sum(x[[product]])
}
total_sales
}
# Display the total monthly sales for each product
f_for(product_sales)

## product1 product2 product3
##      1990      3075      1470
```

Problem 2: Repeat 1 using map.

```
# Load purrr package
library(purrr)

## Warning: package 'purrr' was built under R version 4.5.1
## Error: package or namespace load failed for 'purrr' in dyn.load(file, DLLpath
= DLLpath, ...):
## unable to load shared object '/usr/local/lib/R/site-library/rlang/libs/rlang.so':
## /usr/local/lib/R/site-library/rlang/libs/rlang.so: undefined symbol:
Rf_charIsUTF8

# Use map to calculate total monthly sales for each product
f_map = function(x){
```

```
  map(x, sum)
}
# Display the total monthly sales for each product
f_map(product_sales)

## Error in map(x, sum): could not find function "map"
```

As a result, we obtain a list of three lists. If we want to specify the class of the output, we can use functions such as `map_dbl`, `map_int`, `map_lgl`, `map_chr`, etc.

Problem 3: Repeat 1 using `lapply`.

```
# Use lapply to calculate total monthly sales for each product
f_lapply = function(x){
  lapply(x, sum)
}

# Display the total monthly sales for each product
f_lapply(product_sales)

## $product1
## [1] 1990
##
## $product2
## [1] 3075
##
## $product3
## [1] 1470
```

We observe the same result. Compared to `map`, the function `lapply` is part of base R and always returns a list.

Problem 4: Repeat 1 using `sapply`.

```
# Use sapply to calculate total monthly sales for each product
f_sapply = function(x){
  sapply(x, sum)
}

# Display the total monthly sales for each product
f_sapply(product_sales)

## product1 product2 product3
##      1990      3075      1470
```

As a result, we have a numeric vector.

Problem 5: Repeat 1 using vapply.

```
#5. Vapply

# Use vapply to calculate total monthly sales for each product
f_vapply = function(x){
  vapply(x, FUN = sum, FUN.VALUE = numeric(1))
}

# Display the total monthly sales for each product
f_vapply(product_sales)

## product1 product2 product3
##      1990      3075      1470
```

The function `vapply` produces the same result as `sapply`, but with stricter control over outputs. Unlike `sapply`, which tends to simplify outputs automatically, `vapply` consistently returns the output type specified in `FUN.VALUE`.

Problem 6: Repeat 1 using mclapply or parLapply.

```
#6. Mclapply and parLapply

# We install a library parallel for parallel calculus

library(parallel)

# One way to implement parallelism is to use mclapply (does not
  supported by Windows!)

# mclapply(product_sales, sum, mc.cores = 5)

# Alternatively, one can use parLapply

# To this end, we create 5 clusters
cl <- makeCluster(5)
f_par = function(x){
  # and we are able to apply our function
  parLapply(cl, x, sum)
  # We should stop clusters
}

# Display the total monthly sales for each product

f_par(product_sales)

stopCluster(cl)
```

The advantages of the `mclapply` function:

1. It provides a quick and simple parallel solution without inter-process communication.
2. It is well-suited for tasks that can be completed independently on a single machine.

However, `mclapply` is not supported on Windows (i.e., it is not portable) and does not allow communication between parallel processes.

On the other hand, the function `parLapply` is supported on both Windows and Unix-like systems, and it provides the user with more control over the processes (including parallel computation across multiple computers). To use `parLapply`, however, one needs to create clusters and manage the processes accordingly.

Problem 7: Compare these six approaches with `microbenchmark`. Which approach is the most efficient?

To be able to treat `parLapply` as a function we move `stopCluster(cl)` in the end of the code.

```
# To this end, we create 5 clusters
cl <- makeCluster(5)
f_par = function(x){

  # and we are able to apply our function
  parLapply(cl, x, sum)
  # We should stop clusters
}

# Display the total monthly sales for each product
f_par(product_sales)

# 7. Benchmark

library(microbenchmark)

# Testing performance of the aforementioned functions
microbenchmark(f_for(product_sales), f_map(product_sales), f_
  lapply(product_sales), f_sapply(product_sales), f_vapply(
    product_sales), f_par(product_sales), times = 1000)

stopCluster(cl)
```

The table with results should look as follows:

```
Unit: microseconds
      expr    min      lq     mean  median      uq     max neval cld
f_for(product_sales)  3.5    5.4   7.5209    7.60    8.70   34.2  1000 a
f_map(product_sales) 114.4  138.2 166.6000  159.75  177.60 1193.2  1000 b
f_lapply(product_sales)  2.9    3.8   5.8297    4.50    5.30   971.8  1000 a
f_sapply(product_sales) 13.4   17.6  24.9964   23.30   26.00 1770.9  1000 a
f_vapply(product_sales)  3.6    4.9   7.6922    6.20    7.40 1260.2  1000 a
f_par(product_sales) 524.9  608.5 794.8039  689.35  850.45 8159.1  1000 c
```

According to the table, we can conclude that `lapply` and `for` loop implementation work faster than other functions. The function `parlapply` is the slowest one in this example, which might be a case for simple tasks, since `parlapply` requires time to manage several processes and, therefore, be inefficient in simple problems.