Budowa pakietów z użyciem pakietu devtools

Przemysław Biecek

5 XII 2015

Dane do pracy

W poniższych przykładach będziemy wykorzystywać zbiór danych `auta2012` z pakietu `PogromcyDanych`.

```
library(PogromcyDanych)
auta2012[1:3, 1:5]
```

```
##    Cena Waluta Cena.w.PLN Brutto.netto  KM
## 1 49900    PLN      49900       brutto 140
## 2 88000    PLN      88000       brutto 156
## 3 86000    PLN      86000       brutto 150
```

Zróbmy sobie pakiet!

W programie R zbiory danych i funkcje są grupowane w pakietach. Ułatwia to dystrybucję i zarządzanie powiązanymi ze sobą funkcjonalnościami.

Pakiet buduje się często po to by:

- udostępnić opracowane rozwiązanie, algorytm, funkcjonalność współpracownikom lub reszcie świata,
- ułatwić dostęp do kilku zbiorów danych studentom, współpracownikom, reszcie świata.

Ale warto tworzyć pakiety wokół realizowanych projektów, nawet jeżeli nie mamy w planach nigdy z nikim się tym pakietem podzielić.

Dlaczego? Podstawową funkcją pakietów jest bowiem porządkowanie kodu, danych, dokumentacji i testów. *Korzystanie z pakietów wymusza strukturę* i dlatego warto każdy projekt realizować jako pakiet. Dobrym pomysłem jest też umieszczanie większych pakietów na GitHubie.

Struktura

Pakiety mają wspólną strukturę katalogów. Pakiet to katalog, wewnątrz którego znajdują się:

- plik `DESCRIPTION`, z opisem podstawowych informacji o pakiecie (nazwa, opis, autor, zależności),
- plik `NAMESPACE`, z opisem funkcji udostępnionych przez pakiet,

Cytując Hilary Parker: *Seriously, it doesn't have to be about sharing your code (although that is an added benefit!). It is about saving yourself time.*

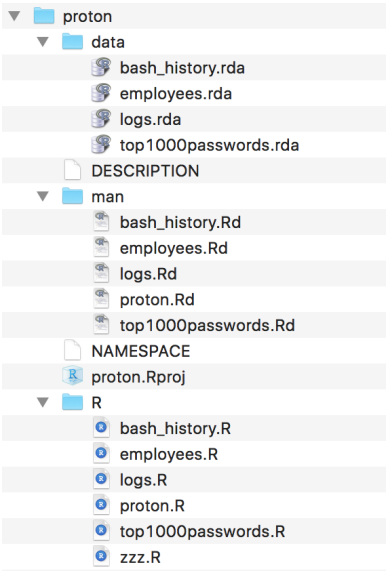


Figure 1: Przykładowa struktura pakietu proton.

- katalog `R`, z listą funkcji w programie R,
- katalog `data`, z listą zbiorów danych udostępnionych przez pakiet,
- katalog `man`, z plikami dokumentacji Rd dla udostępnionych funkcji i zbiorów danych,
- katalog `vignettes`, z listą przewodników opisujących funkcjonalności pakietu,
- katalog `tests`, z listą testów weryfikujących poprawność funkcji.

Tworzenie nowego pakietu

Pakiet to katalog mający określoną strukturę. Można go więc zbudować tworząc w dowolny sposób potrzebne podkatalogi i pliki.

Znacznie wygodniej jest jednak do tego celu wykorzystać pakiet `devtools` i udostępnioną przez niego funkcję `create`.

Stwórzmy w roboczym katalogu nowy pakiet o nazwie `kupPanAuto`.

Przedstawimy budowę pakietów na podstawie zbioru danych auta2012 z pakietu PogromcyDanych

Aby zainstalować pakiet devtools, powinniśmy wcześniej zainstalować go instrukcją install.packages(devtools)

```
library(devtools)
create("kupPanAuto")
```

```
## Creating package kupPanAuto in .
## No DESCRIPTION found. Creating with values:

## Package: kupPanAuto
## Title: What the Package Does (one line, title case)
## Version: 0.0.0.9000
## Authors@R: person("First", "Last", email = "first.last@example.com", role = c("aut", "cre"))
## Description: What the package does (one paragraph).
## Depends: R (>= 3.2.2)
## License: What license is it under?
## LazyData: true

## Adding RStudio project file to kupPanAuto
```

Stworzenie pakietu funkcją `create` powoduje utworzenie pliku z projektem `.Rproj` dla RStudio. Otwórzmy ten projekt, dzięki czemu katalog z pakietem stanie się katalogiem roboczym.

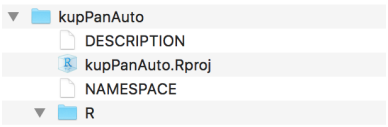


Figure 2: Struktura nowo utworzonego pakietu.

Plik DESCRIPTION

Nowo stworzony pakiet ma generyczny plik DESCRIPTION. Pierwszą rzeczą, którą należy zrobić po zbudowaniu pakietu to uzupełnienie informacji w tym pliku. Większość punktów ma nazwy jednoznacznie określające co powinno być w nie wpisane.

Szczególnie istotne są sekcje `Imports` i `Suggests`.

- Sekcja `Imports` wskazuje pakiety niezbędne do działania Twojego pakietu. R będzie je instalował przy instalacji Twojego pakietu.

- Sekcja `Suggests` wskazuje pakiety dodatkowe, które mogą rozszerzać możliwości Twojego pakietu, ale nie są niezbędne do działania kluczowych funkcjonalności.

```
Package: proton
Title: The Proton Game
Version: 1.0
Authors@R: c(
    person("Przemysław", "Biecek", email = "przemyslaw.biecek@gmail.com", role = c("aut", "cre")),
    person("Witold", "Chodor", email = "witoldchodor@gmail.com", role = c("trl")),
    person("Foundation", "SmarterPoland.pl", email = "smarterpoland@gmail.com", role = c("cph")))
Description: 'The Proton Game' is a console-based data-crunching game for younger and older data sci
  Act as a data-hacker and find Slawomir Pietraszko's credentials to the Proton server.
  You have to solve four data-based puzzles to find the login and password.
  There are many ways to solve these puzzles. You may use loops, data filtering, ordering, aggregati
  Only basics knowledge of R is required to play the game, yet the more functions you know, the more
  The knowledge of dplyr is not required but may be very helpful.
  This game is linked with the ,,Pietraszko's Cave'' story available at http://biecek.pl/BetaBit/War
  It's a part of Beta and Bit series.
  You will find more about the Beta and Bit series at http://biecek.pl/BetaBit.
Depends:
    R (>= 3.0.0)
Imports:
    digest
License: GPL-2
LazyData: true
Encoding: UTF-8
RoxygenNote: 5.0.1
```

Figure 3: Przykładowa treść pliku DESCRIPTION dla pakietu proton

Zadanie 1

Stwórz na swoim komputerze pakiet o nazwie `kupPanAuto`. Zrobimy ten pakiet po to by ułatwić sobie odpowiedź na pytanie ile kosztują używane auta.

Uzupełnij plik DESCRIPTION w tym pakiecie, dodaj swoje dane, opis pakietu oraz informację o licencji.

Cykl życia pakietu

Cykl życia pakietu składa się z kilku kroków:

- Pakiet dostępny jako katalog na dysku
- Pakiet w postaci źródłowej `tar.gz` lub binarnej `zip/tgz`
- Pakiet zainstalowany (pakiet binarny rozpakowany w katalogu z bibliotekami R)
- Pakiet włączony w aktualnej sesji R.

Budowanie pakietu

Aby przejść do postaci źródłowej można wykorzystać polecenie `build`.

Parametrem funkcji build() jest katalog w którym znajduje się pakiet do zbudowania. Domyślnie jest to katalog roboczy.

```
build("kupPanAuto")
```

```
## [1] "/Users/pbiecek/Dropbox/_Prezentacje_I_Szkolenia_/2015 Politechnika Slaska Gliwice/1_budowa_pakieto
```

Instalacja pakietu

Pakiety instalowane są w katalogu z bibliotekami R. Ścieżkę do tych katalogów można odczytać używając funkcji `.libPaths()`.

```
.libPaths()
```

```
## [1] "/Library/Frameworks/R.framework/Versions/3.2/Resources/library"
```

Aby zainstalować pakiet z pliku źródłowego lub z katalogu, można wykorzystać polecenie `install`.

Parametrem funkcji install() jest katalog w którym znajduje się pakiet do zbudowania. Domyślnie jest to katalog roboczy.

```
install("kupPanAuto")
```

Pakiety można instalować z repozytorium CRAN poleceniem `install.packages()` a z repozytoriów GitHub poleceniem `install_github()`.

Poniższa instrukcja zainstaluje pakiet `proton` ze strony `https://github.com/BetaAndBit/PieczaraPietraszki`

```
install_github("BetaAndBit/PieczaraPietraszki/protonENG")
```

Włączanie pakietu

Aby skorzystać z danych lub funkcji pakietu należy pakiet włączyć poleceniem `require()` lub `library()`. Ewentualnie można wykorzystywać operator dostępu `::`.

Lub ::: dla funkcji, które nie są publiczne

```
library(kupPanAuto)
```

Uzupełniamy pakiet

Udało nam się zainstalować pusty pakiet. Uzupełnijmy go teraz interesującą treścią.

Dodajemy zależne pakiety

Jeżeli w naszym pakiecie, chcemy korzystać z funkcji lub danych z innych pakietów, powinniśmy dodać odpowiednią zależność do pliku `DESCRIPTION`.

Możemy w tym celu wykorzystać funkcję `use_package()`, która dodaje informacje o zależnościach od wskazanych pakietach.

```
use_package(package = "PogromcyDanych", pkg = "kupPanAuto")

## Adding PogromcyDanych to Imports
## Refer to functions with PogromcyDanych::fun()

use_package(package = "dplyr", pkg = "kupPanAuto")

## Adding dplyr to Imports
## Refer to functions with dplyr::fun()
```

Dodajemy funkcję

Aby dodać funkcję do pakietu, wystarczy jej definicję dodać w pliku o rozszerzeniu R do katalogu o nazwie `R`. Podczas ładowania pakietu do przestrzeni nazw pakietu wczytywane są wszystkie instrukcje z plików `*.R` umieszczonych w katalogu `R`.

Nazwa pliku `*.R` nie musi być związana z zawartością, ale zwyczajowo nazwy plików odpowiadają nazwom funkcji, które w danym pliku są zaimplementowane.

Przykładowo, poniżej przedstawiamy definicję funkcji `jakiPrzebieg`, liczącą średni przebieg aut z danego rocznika. Definicję tej funkcji należy wkleić do dowolnego pliku z rozszerzeniem `R` w katalogu o nazwie `R`.

```
jakiPrzebieg <- function(rok = "") {
    selected <- auta2012 %>% filter(Rok.produkcji ==
        rok)
    mean(selected$Przebieg.w.km, na.rm = TRUE)
}
```

Również zwyczajowo w pliku `zzz.R` umieszcza się instrukcje, które mają być wczytane jako ostatnie.

Zadanie 2

Dodaj do pakietu `kupPanAuto` funkcję `jakaCena(marka =, rok =)`, która przyjmie dwa argumenty `marka` i `rok` a następnie jako wynik zwróci średnią cenę aut o marce `marka` wyprodukowanych w roku `rok`.

Zbuduj i zainstaluj pakiet `kupPanAuto`. Wywołaj funkcję `jakaCena()` aby sprawdzić, czy poprawnie działa.

Dodajemy dokumentacje dla funkcji

W programie R dokumentacja dla funkcji i danych przechowywana jest w plikach `Rd` w katalogu `man`.

Wygodniej jednak pracować z dokumentacją, gdy jest ona w tym samym pliku co kod R. Z tego powodu sugerowanym rozwiązaniem do tworzenia dokumentacji jest stosowanie coraz bardziej popularnego pakietu `roxygen2`.

Dokumentacja umieszczona jest w wierszach rozpoczynających się od znaków `#'`. Pierwsza linia dokumentacji określa tytuł, kolejny akapit to krótki opis. Następne akapity to rozszerzony opis.

W dokumentacji wykorzystywane są tagi `roxygen2`, rozpoczynające się od znaku `@`.

Najczęściej stosowane tagi to

- `@param` - opis dla określonego parametru opisywanej funkcji
- `@return` - opis dla wyniku funkcji
- `@export` - tag określający, że dana funkcja ma być udostępniona przez pakiet
- `@examples` - blok z przykładami
- `@rdname` - określa nazwę pliku Rd

```
#' @title The Proton Game
#'
#' @description
#' The \code{proton} function is used for solving problems in the data-based game ,,The Proton Game''.
#' Solve four data-based puzzles in order to crack into Pietraszko's account!
#'
#' @param ... \code{proton} function is called by different arguments, which vary depending
#' on a problem that Bit is trying to solve. See \code{Details} in order to learn more about the list of possible
#'
#' @details Every time when some additional hints are needed one should add
#' \code{hint=TRUE} argument to the \code{proton} function.
#'
#' In order to get more information about a user on the Proton server
#' one should pass \code{action = "login"}, \code{login="XYZ"} arguments
#' to the \code{proton} function.
#'
#' @author
#' Przemyslaw Biecek, \email{przemyslaw.biecek@@gmail.com}, SmarterPoland.pl Foundation.
#'
#' @examples
#' \dontrun{
#' proton()
#' proton(hint=TRUE)
#' }
#' @rdname proton
#' @importFrom digest digest
#' @export
```

Figure 4: Przykładowa dokumentacja dla funkcji proton()

Dodajemy dane

W pakietach możemy również przechowywać dane.

Aby dodać zbiór danych do pakietu, wystarczy zapisać go z użyciem funkcji `save()` z rozszerzeniem `rda` lub `RData` w katalogu `data`.

Inna możliwość to użycie funkcji `use_data`.

```
library(PogromcyDanych)
use_data(auta2012, pkg = "kupPanAuto")
```

Dla danych, również możemy dodać dokumentację w formacie `roxygen2`. Poniższa ilustracja przedstawia przykładową dokumentację.

```
#' The history of logs into the Proton server
#'
#' The dataset describing the history of logs: who, from where and when logged into the Proton server.
#' The subsequent columns in this dataset describe:
#' \itemize{
#'   \item login. The login of the user which logs into the Proton server.
#'   \item host. The IP address of the computer, from which the log into the Proton server was detected.
#'   \item date. The date of log into the Proton server. Rows are sorted by this column.
#' }
#'
#' @docType data
#' @keywords datasets
#' @name logs
#' @usage data(logs)
#' @format a data frame with 59366 rows and 3 columns.
NULL

```

Figure 5: Dokumentacja zbioru z danymi

Budowanie dokumentacji

Dodanie opisów dla zbiorów danych czy funkcji w pikach R to jeszcze nie koniec tworzenia dokumentacji. Należy ją teraz przeformatować z postaci Roxygen do plików `Rd` zrozumiałych przez program R.

W tym celu można wykorzystać funkcję `document()`, która generuje pliki `Rd` na podstawie dokumentacji z plików `R`.

```
document("kupPanAuto")
```

```
## Updating kupPanAuto documentation
## Loading kupPanAuto
## First time using roxygen2 4.0. Upgrading automatically...
```

Po wykonaniu tej instrukcji odpowiednie pliki `Rd` są dodawane do katalogu `man`.

Ilustracje

Dokumentacja w formacie `Rd` dotyczy pojedynczych funkcji lub zbiorów danych.

Aby do pakietu dodać szerszy opis, przykłady użycia, ilustracje działania pakietu, w tym celu można wykorzystać winietki / ilustracje.

Dodać ilustrację do pakietu można poleceniem `use_vignette`.

```
use_vignette("Jak kupić auto", pkg = "kupPanAuto")
```

Testy

Prawdziwym błędem jest błąd popełnić i nie naprawić go., Konfucjusz

Bardzo ważnym i bardzo zaniedbywaną częścią tworzenia pakietów jest przygotowywanie testów weryfikujących czy funkcjonalności wciąż działają.

Dla programu R przygotowano kilka różnych rozwiązań pozwalających na integracje testów dla kodu.

W przypadku budowania pakietów, lekkim i wygodnym rozwiązaniem jest korzystanie z testów z pomocą pakietu `testthat`.

Dodajemy testy

Testy dodajemy w plikach `R` do katalogu `tests/testthat`.

Aby dodać ten katalog do naszego pakietu, wraz z potrzebnymi zależnościami, wystarczy wykorzystać instrukcję `use_testthat()`. Przygotuje ona całą niezbędną infrastrukturę.

```
use_testthat("kupPanAuto")
```

Testy to wywołania funkcji `test_that()`, która przyjmuje dwa argumenty.

- Pierwszym jest nazwa testu, gdyby test wykrył błąd, ta nazwa ułatwi określenie co nie działa.
- Drugi parametr to lista oczekiwań, czyli funkcji rozpoczynających się od `expect_...`. Funkcje te sprawdzają, czy wywołanie funkcji kończy się oczekiwanym rezultatem, wartością, błędem, ostrzeżeniem, napisem itp.

```
test_that("Result is a number", {
    expect_true(is.numeric(jakaCena("Kia", 2010)))
})
test_that("An invalid number of parameters", {
    expect_error(jakaCena())
    expect_error(jakaCena("Kia"))
})
test_that("Results are correct", {
    expect_equal(round(jakaCena("Kia", 2010)),
        44544)
```

```
    expect_equal(round(jakaCena("Kia", 2011)),
        65989)
})
```

Uruchamiamy testy

Aby wywołać wszystkie testy wystarczy wywołać funkcję `test()`.

```
library(testthat)
test("kupPanAuto")
```

Zadanie 3

Przygotuj trzy testy dla przygotowanej funkcji. Niech sprawdzają, czy funkcja w poprawny sposób wyznacza wartość dla określonych parametrów, czy poprawnie reaguje na błędy i czy zgadza się klasa wyniku funkcji.

Testujemy pakiet

Aby pakiet został przyjęty na CRAN nie wystarczy, że funkcje działają poprawnie.

Pakiet musi spełniać listę określonych warunków dotyczącej formatowania (Camel Case dla tytułów, niezbyt długie linie, poprawna dokumentacja, wykonywalne przykłady), czy spójności opisu.

Do weryfikacji, czy pakiet jest zgodny z wszystkimi wyznacznikami można wykorzystać funkcję `check()`.

```
check("kupPanAuto")
```

Automatyczne formatowanie kodu

Dla czytelności kodu bardzo ważne jest formatowanie kodu.

Jeżeli nie mamy silnie rozwiniętych nawyków dotyczących formatowania kodu, warto skorzystać z automatycznego formatowania dla przygotowanych funkcji.

Funkcja `tidy_dir()` z pakietu `formatR` formatuje wszystkie pliki R ze wskazanego katalogu.

```
library("formatR")
tidy_dir("kupPanAuto/R")
```

Inne materiały

Streszczeniem materiału o pakietach jest ściągawka pakietu devtools opracowana przez RStudio dostępna na stronie cheatsheet. [1]

[1] https://www.rstudio.com/wp-content/uploads/2015/03/devtools-cheatsheet.pdf

Package Development

with devtools Cheat Sheet

RStudio

Package Structure

A package is a convention for organizing files into directories.

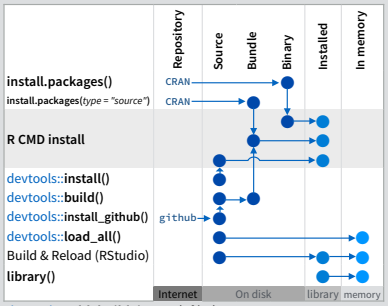
This sheet shows how to work with the 7 most common parts of an R package:

```
Package
  DESCRIPTION   Setup
  R/            Write code
  tests/        Test
  man/          Document
  vignettes/    Teach
  data/         Add data
  NAMESPACE     Organize
```

The contents of a package can be stored on disk as a:

- **source** - a directory with sub-directories (as above)
- **bundle** - a single compressed file (*.tar.gz*)
- **binary** - a single compressed file optimized for a specific OS

Or installed into an R library (loaded into memory during an R session) or archived online in a repository. Use the functions below to move between these states.



devtools::**add_build_ignore**(*"file"*)
Adds file to .Rbuildignore, a list of files that will not be included when package is built.

Setup (DESCRIPTION)

The DESCRIPTION file describes your work and sets up how your package will work with other packages.

You must have a DESCRIPTION file

Add the packages that yours relies on with

devtools::**use_package()**
Adds a package to the Imports field (or Suggests field (if second argument is "Suggests").

CC0	MIT	GPL-2
No strings attached.	MIT license applies to your code if re-shared.	GPL-2 license applies to your code, *and all code anyone bundles with it*, if re-shared.

```
Package: mypackage
Title: Title of Package
Version: 0.1.0
Authors@R: person("Hadley", "Wickham", email =
    "hadley@me.com", role = c("aut", "cre"))
Description: What the package does (one paragraph)
Depends: R (>= 3.1.0)
License: GPL-2
LazyData: true
Imports:
    dplyr (>= 0.4.0),
    ggvis (>= 0.2)
Suggests:
    knitr (>= 0.1.0)
```

Import packages that your package *must have* to work. R will install them when it installs your package.

Suggest packages that are not very essential to yours. Users can install them manually, or not, as they like.

Write code (R/)

All of the R code in your package goes in R/. A package with just an R/ directory is still a very useful package.

Create a new package project with

devtools::**create**(*"path/to/name"*)
Create a template to develop into a package.

Save your code in R/ as scripts (extension .R)

Workflow

1. Edit your code.
2. Load your code with one of

   devtools::**load_all()**
   Re-loads all *saved* files in R/ into memory.

   Ctrl/Cmd + Shift + L (keyboard shortcut)
   Saves all open files then calls load_all().
3. Experiment in the console.
4. Repeat.

- Use consistent style with **r-pkgs.had.co.nz/r.html#style**
- Click on a function and press **F2** to open its definition
- Search for a function with **Ctrl + .**

Visit **r-pkgs.had.co.nz** for more

Test (tests/)

Use tests/ to store unit tests that will inform you if your code ever breaks.

Add a **tests/** directory and import **testthat** with

devtools::**use_testthat()**
Sets up package to use automated tests with testthat

Write tests with **context()**, **test()**, and expectations

Save your tests as .R files in **tests/testthat/**

Workflow

1. Modify your code or tests.
2. Test your code with one of

   devtools::**test()**
   Runs all tests *saved* in tests/.

   Ctrl/Cmd + Shift + T (keyboard shortcut)
3. Repeat until all tests pass

Example test

```
context("Arithmetic")

test_that("Math works", {
  expect_equal(1 + 1, 2)
  expect_equal(1 + 2, 3)
  expect_equal(1 + 3, 4)
})
```

expect_equal()	is equal within small numerical tolerance?
expect_identical()	is exactly equal?
expect_match()	matches specified string or regular expression?
expect_output()	prints specified output?
expect_message()	displays specified message?
expect_warning()	displays specified warning?
expect_error()	throws specified error?
expect_is()	output inherits from certain class?
expect_false()	returns FALSE?
expect_true()	returns TRUE?

Document (man/)

man/ contains the documentation for your functions, the help pages in your package.

Use roxygen comments to document each function beside its definition

Document the name of each exported data set

Include helpful examples for each function

Workflow

1. Add roxygen comments in your .R files
2. Convert roxygen comments into documentation with one of

   devtools::**document()**
   Converts roxygen comments to .Rd files and places them in man/. Builds NAMESPACE.

   Ctrl/Cmd + Shift + D (Keyboard Shortcut)
3. Open help pages with **?** to preview documentation
4. Repeat

.Rd formatting tags

```
\emph{italic text}
\strong{bold text}
\code{function(args)}
\pkg{package}

\dontrun{code}
\dontshow{code}
\donttest{code}

\deqn{a + b (block)}
\eqn{a + b (inline)}

\email{name@@foo.com}
\href{url}{display}
\url{url}

\link[=dest]{display}
\linkS4class{class}
\code{\link{function}}
\code{\link[package]{function}}

\tabular{lcr}{
  left \tab centered \tab right \cr
  cell \tab cell     \tab cell  \cr
}
```

The **roxygen** package

roxygen lets you write documentation inline in your .R files with a shorthand syntax.

- Add roxygen documentation as comment lines that begin with **#'**.
- Place comment lines directly above the code that defines the object documented.
- Place a roxygen **@** tag (right) after **#'** to supply a specific section of documentation.
- Untagged lines will be used to generate a title, description, and details section (in that order)

```
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#' @return The sum of \code{x} and \code{y}.
#' @examples
#' add(1, 1)
#' @export
add <- function(x, y) {
  x + y
}
```

Common roxygen tags

@aliases	@inheritParams	**@seealso**
@concepts	@keywords	@format
@describeIn	**@param**	@source data
@examples	@rdname	@include
@export	**@return**	@slot S4
@family	@section	@field RC

Add data (data/)

The data/ directory allows you to include data with your package.

Store data in one of **data/**, **R/Sysdata.rda**, **inst/extdata**

Always use **LazyData: true** in your DESCRIPTION file.

Save data as .Rdata files (suggested)

devtools::**use_data()**
Adds a data object to data/ (R/Sysdata.rda if **internal = TRUE**)

devtools::**use_data_raw()**
Adds an R Script used to clean a data set to data-raw/. Includes data-raw/ on .Rbuildignore.

Store data in

- **data/** to make data available to package users
- **R/sysdata.rda** to keep data internal for use by your functions.
- **inst/extdata** to make raw data available for loading and parsing examples. Access this data with **system.file()**

Organize (NAMESPACE)

The NAMESPACE file helps you make your package self-contained: it won't interfere with other packages, and other packages won't interfere with it.

Export functions for users by placing **@export** in their roxygen comments

Import objects from other packages with **package::object** (recommended) or **@import**, **@importFrom**, **@importClassesFrom**, **@importMethodsFrom** (not always recommended)

Workflow

1. Modify your code or tests.
2. Document your package (devtools::**document()**)
3. Check NAMESPACE
4. Repeat until NAMESPACE is correct

Teach (vignettes/)

vignettes/ holds documents that teach your users how to solve real problems with your tools.

Create a vignettes/ directory and a template vignette with

devtools::**use_vignette()**
Adds template vignette as vignettes/my-vignette.Rmd.

Append YAML headers to your vignettes (like right)

Write the body of your vignettes in R Markdown (rmarkdown.rstudio.com)

```
---
title: "Vignette Title"
author: "Vignette Author"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

Submit your package

Pozycją obowiązkową jest nowa książka Hadleya o tym jak pisać pakiety. Lektura na 3-4 godziny dostępna na stronie `http://r-pkgs.had.co.nz/intro.html`

Bardzo rozbudowana, niespecjalnie czytelna ale wyczerpująca (z różnymi znaczeniami tego słowa) dokumentacja dotycząca pakietów dostępna jest na stronie „R extensions". [2]

Uzupełnieniem dla obu powyższych jest dokumentacja pakietu `roxygen2` dostępna na stronie projektu. [3]

Więcej o testach i pakiecie testthat przeczytać można w artykule *testthat: Get Started with Testing* Hadley Wickham. [4]

[2] `http://cran.r-project.org/doc/manuals/R-exts.html#Creating-R-packages`

[3] `https://github.com/klutometis/roxygen#roxygen2`

[4] `https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf`

Rozwiązania

```
# Zadanie 2
cat(file="kupPanAuto/R/jakaCena.R", "
jakaCena <- function(marka ='', rok ='') {
   selected <- auta2012 %>%
    filter(Marka == marka, Rok.produkcji == rok)
    mean(selected$Cena.w.PLN, na.rm=TRUE)
}
")
install("kupPanAuto")
library(kupPanAuto)
jakaCena(marka="Kia", rok=2010)

# Zadanie 3
cat(file="kupPanAuto/tests/testthat/test1.R", '
test_that("Result is a number",{
  expect_true(is.numeric(jakaCena("Kia", 2010)))
})
test_that("An invalid number of parameters",{
  expect_error(jakaCena())
  expect_error(jakaCena("Kia"))
})
test_that("Results are correct",{
  expect_equal(round(jakaCena("Kia", 2010)),
               44544)
  expect_equal(round(jakaCena("Kia", 2011)),
               65989)
})
')
test("kupPanAuto")
```