O'REILLY®

Wydanie II

Pakiety R

Zarządzanie, testowanie, dokumentacja i udostępnianie kodu

Hadley Wickham
Jennifer Bryan

Helion

Tytuł oryginału: R Packages: Organize, Test, Document, and Share Your Code, 2nd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-1046-1

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/pakir2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 4

Podstawy sposobu pracy

W poprzednim rozdziale przedstawiliśmy nieco informacji na temat pakietów i bibliotek języka R. Natomiast w tym rozdziale zaprezentujemy podstawowe sposoby pracy podczas tworzenia pakietu i jego przechodzenia między różnymi stanami w trakcie pracy nad nim.

Tworzenie pakietu

Wiele pakietów powstaje na skutek czyjejś frustracji spowodowanej tym, że określone zadanie jest trudne do wykonania. W jaki sposób ustalić, czy dane zadanie jest warte opracowania dla niego pakietu? Na to pytanie nie ma jednoznacznej odpowiedzi. Jednak warto sobie uświadomić przynajmniej dwa typy korzyści:

Produkt

Twoja praca stanie się wygodniejsza, gdy dana funkcjonalność zostanie formalnie zaimplementowana w pakiecie.

Proces

Większe doświadczenie z zakresu R pozwoli na efektywniejszą pracę z tym językiem.

Analiza istniejącego środowiska

Jeżeli zależy Ci tylko na istnieniu produktu, wówczas głównym celem będzie poruszanie się po obszarze istniejących pakietów. Julia Silge, John C. Nash i Spencer Graves przeprowadzili na ten temat ankiety i sesje mniej więcej w czasie konferencji useR! 2017, a przygotowane opracowanie dla „The R Journal" dostarcza wielu cennych zasobów[1].

Jeżeli szukasz sposobów na pogłębienie wiedzy z zakresu R, nie przestawaj poznawać środowiska tego języka. Istnieje wiele dobrych powodów, dla których warto samodzielnie opracować pakiet, nawet jeśli już istnieje pakiet oferujący taką funkcjonalność. Ekspertem można się stać poprzez faktyczne tworzenie rozwiązań, często nawet bardzo prostych. Zasługujesz na tę samą możliwość uczenia się poprzez eksperymentowanie. Jeżeli można byłoby pracować jedynie nad projektami,

[1] Julia Silge, John C. Nash i Spencer Graves, *Navigating the R Package Universe*, „The R Journal", 2018, nr 2(10), s. 558–563 (*https://journal.r-project.org/archive/2018/RJ-2018-058/index.html*).

które nigdy wcześniej nie zostały zrealizowane, prawdopodobnie oznaczałoby to konieczność mierzenia się z problemami niezwykle rzadkimi bądź wyjątkowo trudnymi do rozwiązania.

Przydatność istniejących narzędzi warto również ocenić na podstawie interfejsu użytkownika, wartości domyślnych oraz sposobu działania w przypadkach skrajnych. Formalnie rzecz biorąc, pakiet może być tym, czego potrzebujesz, choć jednocześnie może okazać się bardzo nieergonomicznym rozwiązaniem. W takich przypadkach znacznie sensowniejszym rozwiązaniem będzie opracowanie własnej implementacji lub utworzenie funkcji opakowania, które ułatwią pracę z pakietem.

Jeżeli Twoja praca mieści się w doskonale ugruntowanej dziedzinie, warto dowiedzieć się więcej na temat istniejących pakietów R, nawet jeśli zdecydujesz się na rozwiązanie problemu przez opracowanie własnego pakietu. Czy w pakiecie zostały wykorzystane określone wzorce projektowe? Czy zastosowanie znalazły konkretne struktury danych, które są często spotkane podczas obsługi danych wejściowych i wyjściowych? Na przykład istnieje bardzo aktywna społeczność języka R zgromadzona wokół analizy danych przestrzennych (*https://r-spatial.org/*), której udało się samodzielnie zorganizować się i wypromować większą spójność między pakietami, za które odpowiadają różni opiekunowie. W zakresie modelowania pakiet hardhat (*https://hardhat.tidymodels.org/*) oferuje szkielet przeznaczony do utworzenia pakietu modelowania, który będzie świetnie dopasowany do ekosystemu tidymodels (*https://www.tidymodels.org/*). Jeżeli Twój pakiet elegancko wpasuje się w ekosystem, to stanie się częściej używany i będzie wymagał mniej dokumentacji.

Nadanie nazwy pakietowi

> „Są tylko dwa trudne zadania w informatyce: unieważnienie bufora i nadawanie nazw".
>
> — Phil Karlton

Zanim będzie można przystąpić do utworzenia pakietu, trzeba nadać mu nazwę. Może się to okazać najtrudniejszym zadaniem w trakcie pracy nad pakietem. (Po części dlatego, że nikt Cię w tym nie wyręczy).

Wymagania formalne

Istnieją trzy wymagania formalne w zakresie nadawania nazw:

- Nazwa może składać się jedynie z liter, cyfr i kropek.
- Nazwa musi rozpoczynać się od litery.
- Nazwa nie może kończyć się kropką.

To niestety oznacza, że nie można używać łączników ani znaków podkreślenia w nazwie pakietu. Odradzamy również stosowanie kropek ze względu na możliwość pomylenia z rozszerzeniami plików i metodami systemu S3.

Kwestie warte rozważenia

Jeżeli planujesz współdzielenie pakietu z innymi, bardzo ważne jest nadanie mu dobrej nazwy. Oto kilka związanych z tym wskazówek:

- Zdecyduj się na unikatową nazwę, którą będzie można łatwo znaleźć za pomocą wyszukiwarki internetowej. Dzięki temu potencjalni użytkownicy będą mogli bez problemu znaleźć ten pakiet (i powiązane z nim zasoby), a Ty łatwo sprawdzisz, kto z niego korzysta.
- Nie wybieraj nazwy już użytej w repozytorium CRAN lub Bioconductor. Pod uwagę trzeba wziąć jeszcze inne rodzaje kolizji nazw:
  - Czy aktualnie jest już opracowywany pakiet o takiej samej nazwie (np. w serwisie GitHub istnieje jego historia i wydaje się, że ten pakiet wkrótce zostanie wydany)?
  - Czy dana nazwa jest już używana dla innego oprogramowania, biblioteki bądź frameworka w ekosystemie (np. Pythona bądź JavaScriptu)?
- Unikaj jednoczesnego używania małych i wielkich liter, ponieważ to utrudnia wpisywanie nazwy, a nawet zapamiętanie (np. trudno jest zapamiętać, czy nazwą pakietu było Rgtk2, RGTK2 czy może RGtk2).
- Preferuj nazwy łatwe do wymówienia, aby użytkownicy mogli komfortowo rozmawiać o Twoim pakiecie.
- Znajdź słowo opisujące problem i zmodyfikuj je w taki sposób, aby było unikatowe. Oto kilka przykładów:
  - lubridate ułatwia pracę z datą i godziną;
  - rvest pobiera zawartość stron internetowych;
  - r2d3 dostarcza narzędzia przeznaczone do pracy z wizualizacjami D3;
  - forcats to anagram słowa factors użytego *dla* (ang. *for*) danych kategoryzujących.
- Korzystaj ze skrótów, np.:
  - Rcpp = R + C++ (plus plus);
  - brms = Bayesian Regression Models using Stan.
- Dodaj literę R:
  - stringr dostarcza narzędzia przeznaczone do pracy z ciągami tekstowymi;
  - beepr odtwarza powiadomienia dźwiękowe;
  - callr wywołuje kod R z poziomu języka R.
- Unikaj problemów natury prawnej.
  - Jeżeli tworzysz pakiet współdziałający z usługą komercyjną, sprawdź informacje dotyczące brandingu. Na przykład pakiet rDrop nie został nazwany rDropbox, ponieważ Dropbox nie zezwala, aby jakakolwiek aplikacja używała pełnej nazwy będącej zastrzeżonym znakiem towarowym.

W swoim zabawnym poście opublikowanym na blogu pod adresem *https://www.njtierney.com/post/2018/06/20/naming-things/* Nick Tierney przedstawił typologię nazw pakietów, w której

znajdziesz wiele inspirujących przykładów. Podzielił się również swoimi doświadczeniami dotyczącymi zmiany nazwy pakietu — post „So, you've decided to change your r package name" (*https://www.njtierney.com/post/2017/10/27/change-pkg-name/*) to świetne źródło informacji, jeśli nie udało Ci się zrobić tego dobrze za pierwszym razem.

Użyj dostępnego pakietu

Niemożliwe będzie dostosowanie się do wszystkich wcześniej wymienionych wskazówek, dlatego trzeba zdecydować się na pewien kompromis. Pakiet available (*https://cran.r-project.org/web/packages/available/index.html*) zawiera funkcję o nazwie `available()`, pomocną w przeanalizowaniu pod wieloma względami potencjalnej nazwy pakietu:

```
library(available)

available("doofus")
#> Urban Dictionary can contain potentially offensive results,
#>   should they be included? [Y]es / [N]o:
#> 1: 1
#> ── doofus
──────────────────────────────────────────────────────────────────────────────
#> Name valid:  ✓
#> Available on CRAN:  ✓
#> Available on Bioconductor:  ✓
#> Available on GitHub:  ✓
#> Abbreviations: http://www.abbreviations.com/doofus
#> Wikipedia: https://en.wikipedia.org/wiki/doofus
#> Wiktionary: https://en.wiktionary.org/wiki/doofus
#> Sentiment:???
```

Funkcja `available()` wykonuje kilka zadań:

- Sprawdzenie poprawności nazwy.
- Sprawdzenie dostępności nazwy w repozytoriach takich jak CRAN, Bioconductor itd.
- Sprawdzenie nazwy w różnych witrynach internetowych pod kątem niezamierzonego znaczenia wybranej nazwy. W trakcie sesji interaktywnej adresy URL wyświetlone przez pakiet available pojawią się na kartach przeglądarki WWW.
- Próba oceny wydźwięku nazwy: pozytywny czy negatywny.

Funkcją alternatywną o podobnym przeznaczeniu jest `pak::pkg_name_check()`. Skoro pakiet pak jest obecnie bardziej aktywnie rozwijany niż available, wydaje się lepszym rozwiązaniem na przyszłość.

Tworzenie pakietu

Po określeniu nazwy pakietu można go utworzyć na dwa sposoby:

- Wywołanie funkcji `usethis::create_package()`.
- Użycie opcji menu *File/New Project/New Directory/R Package* w RStudio. W tym przypadku ostatecznie zostanie wywołana funkcja `usethis::create_package()`, więc tak naprawdę istnieje tylko jeden sposób na utworzenie pakietu.

W wyniku tych działań otrzymasz najmniejszy możliwy *działający* pakiet, który będzie składał się z trzech komponentów:

- katalog *R*, na którego temat więcej dowiesz się w rozdziale 6.;
- podstawowa wersja pliku *DESCRIPTION*, na którego temat więcej dowiesz się w rozdziale 9.;
- podstawowa wersja pliku *NAMESPACE*, na którego temat więcej dowiesz się w rozdziale 10.

Pakiet może zawierać jeszcze plik projektu RStudio, *pkgname.Rproj*, dzięki któremu będzie można łatwo go używać w środowisku IDE RStudio, jak to wyjaśniliśmy nieco dalej w tym rozdziale. Znajdziesz również podstawowe wersje plików *.Rbuildignore* i *.gitignore*.

Do utworzenia pakietu nie wykorzystuj wywołania `package.skeleton()`. Skoro ta funkcja pochodzi ze środowiska R, być może kuszące będzie jej użycie. Jednak powoduje ona utworzenie pakietu, który natychmiast zgłasza błędy podczas stosowania polecenia `R CMD build`. Wymieniona funkcja wykorzystuje inne podejście do tworzenia pakietów niż omówione w tej książce. Dlatego naprawa uszkodzonego stanu początkowego oznacza konieczność wykonania dodatkowej pracy przez programistów używających ekosystemu devtools (a zwłaszcza pakietu roxygen2). Do utworzenia pakietu skorzystaj z wywołania `create_package()`.

Dlaczego należy korzystać z wywołania create_package()?

Podstawowym i jedynym argumentem wymaganym przez wywołanie `create_package()` jest ścieżka dostępu wskazująca położenie, w którym ma zostać utworzony nowy pakiet:

```
create_package("path/to/package/pkgname")
```

Pamiętaj, że to będzie miejsce, w którym znajdzie się pakiet w postaci kodu źródłowego (rozdział 3.), a nie pakiet zainstalowany (rozdział 3.). Pakiety zainstalowane trafiają do *biblioteki*, a procedura przygotowania biblioteki została również omówiona w poprzednim rozdziale.

Gdzie należy przechowywać pakiety w postaci kodu źródłowego? Taka lokalizacja powinna być oddzielna od tej, w której znajdą się pakiety zainstalowane. W przypadku braku zewnętrznych uwarunkowań typowy użytkownik powinien na pakiety języka R (w postaci kodu źródłowego) przeznaczyć podkatalog znajdujący się w katalogu domowym. Omawialiśmy tę kwestię z innymi programistami i okazuje się, że kod źródłowy wielu pakietów tidyverse zostaje umieszczony w katalogu *~/rrr/*, *~/documents/tidyverse/*, *~/r/packages/* lub *~/pkg/*. Niektórzy z nas używają do tego pojedynczego katalogu, inni dzielą pakiety w postaci kodu źródłowego na kilka katalogów, w zależności od roli podczas programowania (współautor bądź nie), użycia serwisu GitHub (tidyverse kontra r-lib), etapu programistycznego (aktywny kontra nieaktywny) itd.

Wymienione konwencje dotyczące katalogów odzwierciedlają to, że przede wszystkim zajmujemy się budowaniem narzędzi. Pracownik naukowy może organizować pliki według poszczególnych publikacji, podczas gdy danolog może układać pakiety według produktów danych i raportów. Nie istnieje żaden techniczny ani tradycyjny powód, dla którego miałoby być preferowane konkretne podejście. O ile jesteś w stanie zapewnić wyraźne oddzielenie pakietów w postaci kodu źródłowego od pakietów zainstalowanych, wybierz strategię, która Ci najbardziej odpowiada i wpisuje się w ogólny sposób organizacji plików w systemie. A następnie konsekwentnie ją stosuj.

Projekty RStudio

Pakiet devtools współpracuje ze środowiskiem RStudio, które dla większości użytkowników języka R uważamy za najlepsze środowisko IDE. Chcemy w tym miejscu podkreślić, że można korzystać z ekosystemu devtools bez sięgania po RStudio i na odwrót — możliwe jest tworzenie pakietów R w RStudio bez używania devtools. Jednak istnieje specjalna dwustronna relacja między nimi, dzięki której stosowanie pakietu devtools w połączeniu z RStudio przynosi wiele korzyści.

RStudio

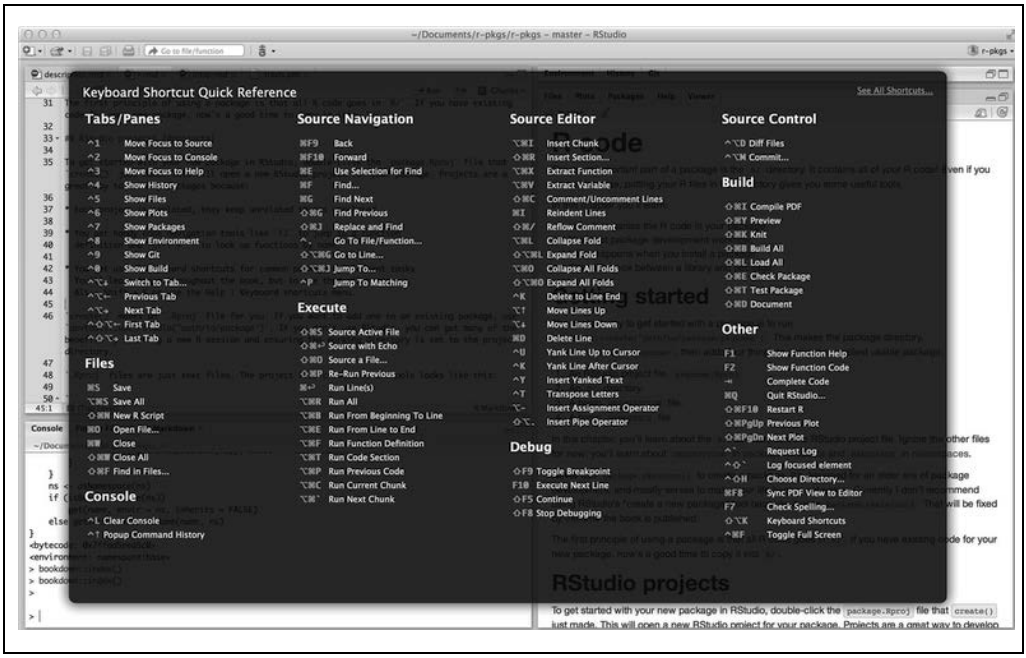
Projekt (słowo zaczynające się od wielkiej litery „P") RStudio to zwykły katalog w komputerze zawierającym pewną (w większości ukrytą) infrastrukturę RStudio ułatwiającą pracę nad jednym lub większą liczbą *projektów* (słowo zaczynające się od małej litery „p"), którymi mogą być np.: pakiet R, analiza danych, elegancka aplikacja, książka, blog.

Zalety używania projektów RStudio

Z lektury poprzedniego rozdziału wiesz, że pakiet w postaci kodu źródłowego znajduje się w katalogu na dysku twardym komputera. Gorąco zachęcamy, aby wszystkie pakiety tego typu były również Projektami RStudio. Oto kilka wybranych zalet takiego rozwiązania:

- Projekt można „uruchomić". Bardzo łatwo można uruchomić nowy egzemplarz RStudio w ramach Projektu, mając do dyspozycji przeglądarkę plików i bieżący katalog roboczy skonfigurowane w dopasowany sposób, gotowe do pracy.
- Poszczególne Projekty są odizolowane — kod uruchomiony w jednym Projekcie nie ma wpływu na żaden inny Projekt.
  - Jednocześnie można mieć wiele otwartych Projektów RStudio i wówczas kod wykonywany w Projekcie A nie będzie miał żadnego wpływu na sesję R oraz przestrzeń roboczą w Projekcie B.
- Otrzymujesz możliwość wygodnej nawigacji po kodzie źródłowym, np. naciśnięcie klawisza *F2* pozwala przejść do definicji funkcji, a naciśnięcie klawiszy *Ctrl+.* spowoduje wyszukiwanie funkcji bądź plików po ich nazwie.
- Dostęp do użytecznych skrótów klawiszowych i interaktywnego interfejsu użytkownika dla najczęściej wykonywanych zadań związanych z tworzeniem pakietów w języku R, takich jak generowanie dokumentacji, wykonywanie testów lub sprawdzanie całego pakietu.

Aby poznać najbardziej użyteczne skróty klawiszowe, naciśnij klawisze *Alt+Shift+K* albo wybierz opcję menu *Help/Keyboard Shortcuts Help*. Zobaczysz informacje podobne do pokazanych na rysunku 4.1.



Rysunek 4.1. Skróty klawiszowe wyświetlone w RStudio

RStudio

RStudio oferuje również tzw. paletę poleceń (*https://docs.posit.co/ide/user/ide/reference/shortcuts.html#command-palette*), która zapewnia szybki i umożliwiający wyszukiwanie dostęp do wszystkich poleceń zintegrowanego środowiska programistycznego. Okazuje się to szczególnie użyteczne, gdy trudno jest zapamiętać określony skrót klawiszowy. Paleta zostaje wyświetlona po naciśnięciu klawiszy *Ctrl+Shift+P* (Windows i Linux) bądź *Cmd+Shift+P* (macOS).

RStudio

Obserwuj zespół RStudio (*https://twitter.com/rstudiotips*) w serwisie Twitter, a znajdziesz tam regularną porcję wskazówek i podpowiedzi.

Tworzenie Projektu RStudio

Jeżeli stosujesz się do naszego zalecenia, aby nowy pakiet tworzyć za pomocą wywołania `create_package()`, to każdy nowy pakiet będzie również Projektem RStudio, o ile pracujesz w środowisku RStudio.

Gdy zachodzi potrzeba wskazania istniejącego katalogu kodu źródłowego pakietu jako Projektu RStudio, wybierz jedną z następujących możliwości.

- Z poziomu RStudio wybierz opcję menu *File/New Project/Existing Directory*.
- Użyj wywołania `create_package()` razem ze ścieżką dostępu do istniejącego katalogu z kodem źródłowym pakietu R.
- Użyj wywołania `usethis::use_rstudio()` razem z aktywnym projektem usethis (więcej informacji na ten temat znajdziesz nieco dalej w tym rozdziale) utworzonym na podstawie istniejącego pakietu R. W praktyce to oznacza, że bieżący katalog roboczy musi się znajdować w istniejącym katalogu pakietu.

Co tworzy Projekt RStudio?

Katalog będący Projektem RStudio będzie zawierał plik o nazwie *.Rproj*. Zwykle jeśli nazwą katalogu jest *foo*, plik Projektu będzie miał nazwę *foo.Rproj*. Jeżeli dany katalog jest również pakietem R, nazwą pakietu przeważnie też będzie *foo*. Takie podejście ma zapewnić zbieżność wszystkich tych nazw i *nie* prowadzić do zagnieżdżania pakietu w podkatalogu Projektu. Jeżeli zdecydujesz się na zastosowanie innego sposobu pracy, może to oznaczać, że będziesz się zmagać z narzędziami.

Plik *.Rproj* to zwykły plik tekstowy. Spójrz na jego przykładową zawartość w Projekcie utworzonym za pomocą usethis:

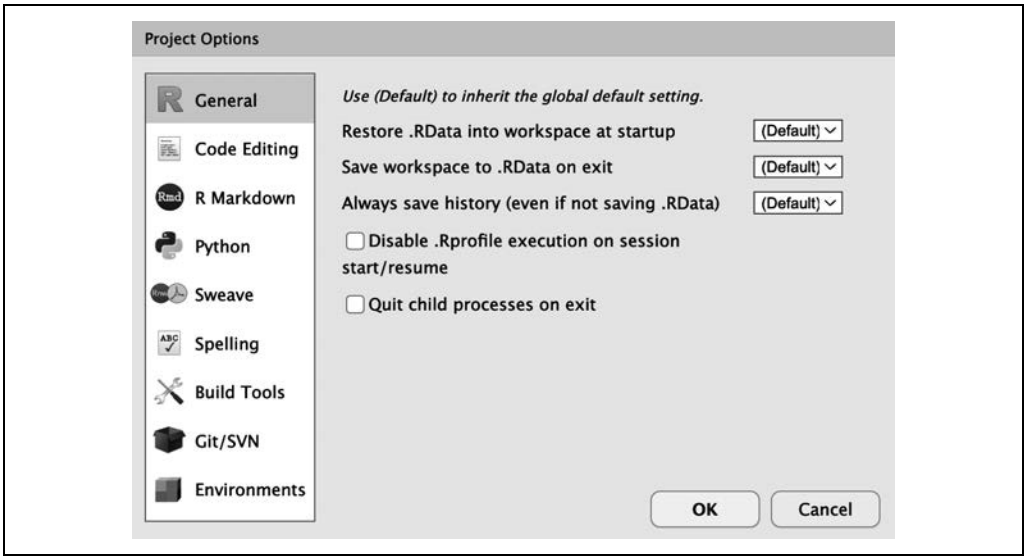
```
Version: 1.0

RestoreWorkspace: No
SaveWorkspace: No
AlwaysSaveHistory: Default

EnableCodeIndexing: Yes
Encoding: UTF-8

AutoAppendNewline: Yes
StripTrailingWhitespace: Yes
LineEndingConversion: Posix

BuildType: Package
PackageUseDevtools: Yes
PackageInstallArgs: --no-multiarch --with-keep.source
PackageRoxygenize: rd,collate,namespace
```

Nie musisz ręcznie modyfikować jego zawartości. Zamiast tego skorzystaj z funkcjonalności środowiska RStudio dostępnej po wybraniu opcji menu *Tools/Project Options* (rysunek 4.2) lub *Project Options* w menu Projektu, w prawym górnym rogu okna RStudio (rysunek 4.3).

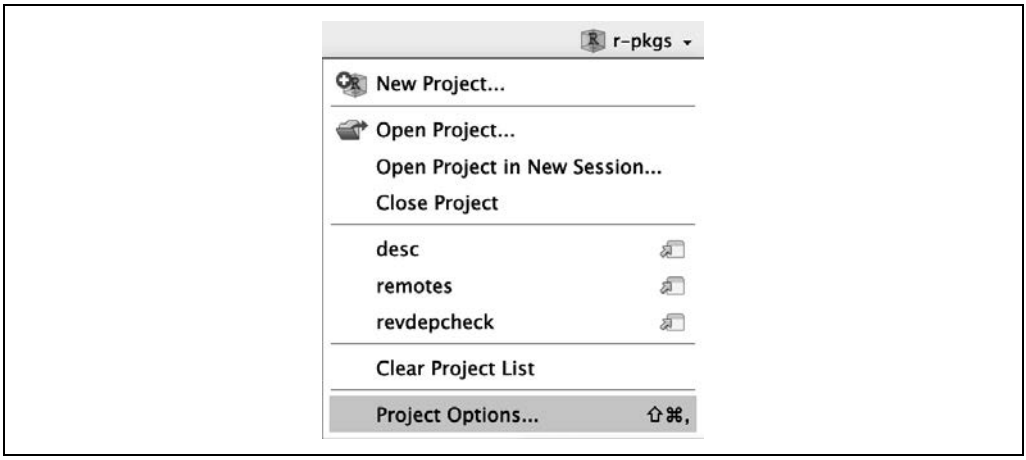
Uruchamianie Projektu RStudio

Dwukrotne kliknięcie pliku *foo.Rproj* w Finderze systemu macOS albo w Eksploratorze systemu Windows spowoduje uruchomienie Projektu w RStudio.

Do tego celu można również wykorzystać opcję menu *File > Open Project (in New Session)* lub menu Projektu w prawym górnym rogu okna RStudio.



Rysunek 4.2. Opcje Projektu w RStudio



Rysunek 4.3. Menu Projektu w RStudio

Jeżeli korzystasz z aplikacji przeznaczonej do uruchamiania programów, prawdopodobnie możesz ją skonfigurować także do obsługi plików *.Rproj*. Oboje używamy do tego Alfreda[2], czyli aplikacji dostępnej jedynie na platformie macOS, ale podobne narzędzia istnieją również dla Windowsa. W rzeczywistości to jest bardzo dobry powód do używania tego rodzaju aplikacji.

Całkowicie normalne — i produktywne — jest mieć otwartych jednocześnie wiele Projektów RStudio.

[2] Narzędzie Alfred zostało skonfigurowane w taki sposób, aby pliki *.Rproj* umieszczało w wynikach wyszukiwania, gdy proponuje listę aplikacji lub plików do otworzenia. Jeżeli chcesz zarejestrować typ pliku *.Rproj* jako obsługiwany przez Alfreda, wybierz w tym narzędziu opcję menu *Preferences/Features/Default Results/Advanced*, przeciągnij dowolny plik *.Rproj* na okno, a następnie je zamknij.

Projekt RStudio kontra aktywny projekt usethis

Zwróć uwagę na to, że większość funkcji usethis nie pobiera ścieżki dostępu i operuje na plikach znajdujących się w „aktywnym projekcie usethis". Przyjmuje się założenie, że w 95% przypadków pliki znajdują się w:

- bieżącym Projekcie RStudio, o ile używane jest to środowisko;
- aktywnym projekcie usethis;
- bieżącym katalogu roboczym dla procesu R.

Jeżeli coś wydaje się dziwne, skorzystaj z wywołania `proj_sitrep()` w celu otrzymania „raportu o sytuacji". To pozwoli zidentyfikować sytuacje szczególne i zaproponować rozwiązanie umożliwiające powrót do stanu normalnego.

```
# these should usually be the same (or unset)
proj_sitrep()
#> *   working_directory: '/Users/jenny/rrr/readxl'
#> * active_usethis_proj: '/Users/jenny/rrr/readxl'
#> * active_rstudio_proj: '/Users/jenny/rrr/readxl'
```

Bieżący katalog roboczy i dyscyplina związana ze ścieżkami dostępu

Podczas tworzenia pakietu będziesz wykonywać kod w języku R. To będzie połączenie wywołań typu `document()` i `test()` z wywołaniami doraźnymi, które pomagają w opracowywaniu funkcji, przykładów i testów. *Gorąco zachęcamy* do tego, aby najwyższego poziomu katalog pakietu w postaci kodu źródłowego był katalogiem roboczym procesu R. Zwykle tak się dzieje domyślnie, więc tak naprawdę jest to zalecenie unikania procesów programistycznych, które będą wymagały majstrowania przy bieżącym katalogu roboczym.

Jeżeli nie masz żadnego doświadczenia w tworzeniu pakietów R, to nie masz również zbyt wielu powodów, aby stosować lub nie stosować się do tego zalecenia. Natomiast doświadczeni programiści R mogą to zalecenie uznać za niepokojące. Być może zastanawiasz się, jak wyrazić ścieżki dostępu podczas pracy w podkatalogach, np. *tests*. Gdy pojawi się taka konieczność, wyjaśnimy, jak skorzystać z wbudowanych procedur pomocniczych, takich jak `testthat::test_path()`, które w trakcie wykonywania będą określały ścieżki dostępu.

Podstawowa idea polega na tym, że jeśli korzystasz jedynie z katalogu roboczego, to stanowi to zachętę do stosowania ścieżek dostępu, które wyraźnie wskazują intencje („odczytaj plik *foo.csv* z katalogu *test*") zamiast niejawnie („odczytaj plik *foo.csv* z bieżącego katalogu roboczego, którym, *jak sądzę*, będzie katalog *test*"). Pewną oznaką polegania na niejawnych ścieżkach dostępu jest nieustanne majstrowanie przy katalogu roboczym, ponieważ używasz funkcji `setwd()` w celu ręcznego spełnienia ukrytych założeń w ścieżkach dostępu.

Korzystanie z jawnie zdefiniowanych ścieżek dostępu może uchronić przed całą gamą problemów i jednocześnie zagwarantować, że programowanie będzie przyjemnością. Mamy dwa powody, dla których trudno jest poprawnie korzystać z niejawnie wyrażonych ścieżek dostępu:

- Jak zapewne pamiętasz, w trakcie pracy nad pakietem R może on znajdować się w różnych postaciach (rozdział 3.). Różnice między poszczególnymi stanami dotyczą m.in. istniejących w nich plików i katalogów oraz ich względnego położenia w hierarchii. Trudne będzie tworzenie względnych ścieżek dostępu funkcjonujących we wszystkich stanach projektu.
- Ostatecznie pakiet zostanie przez Ciebie i potencjalnie także przez repozytorium CRAN przetworzony za pomocą wbudowanych narzędzi typu `R CMD build`, `R CMD check` i `R CMD INSTALL`. Trudne będzie śledzenie, który katalog jest bieżącym katalogiem roboczym na poszczególnych etapach procesu.

Funkcje pomocnicze związane ze ścieżkami dostępu, takie jak `testthat::test_path()` i `fs::path_package()`, a także pakiet rprojroot (*https://rprojroot.r-lib.org/*) okazują się niezwykle użyteczne podczas tworzenia niezawodnych ścieżek dostępu, które będą radziły sobie w różnych sytuacjach pojawiających się w trakcie tworzenia i używania pakietów. Innym sposobem na wyeliminowanie zawodnych ścieżek dostępu jest rygorystyczne stosowanie poprawnych metod związanych z przechowywaniem danych w pakiecie (rozdział 7.) oraz użycie katalogu tymczasowego sesji, gdy to będzie stosowne, np. podczas testowania ulotnych artefaktów (rozdział 13.).

Funkcja load_all()

Nie ulega wątpliwości, że funkcja `load_all()` zalicza się do najważniejszych w ekosystemie devtools:

```
# with devtools attached and
# working directory set to top-level of your source package ...

load_all()

# ... now experiment with the functions in your package
```

Ta funkcja ma znaczenie kluczowe w stosowanym tutaj cyklu tworzenia pakietów w języku R:

1. Dopracowanie definicji funkcji.
2. Wywołanie `load_all()`.
3. Wypróbowanie zmiany przez uruchomienie małego przykładu lub przeprowadzenie testów.

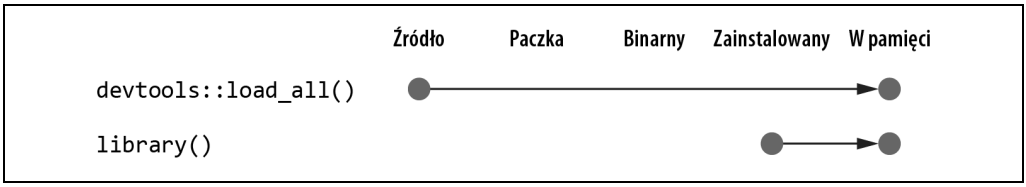
Jeżeli dopiero zaczynasz tworzenie pakietów w R bądź pracę z pakietem devtools, możesz nie dostrzegać wagi funkcji `load_all()` i wyrobić sobie dziwne nawyki znane ze sposobów pracy stosowanych podczas analizy danych.

Zalety funkcji load_all()

Gdy rozpoczynasz korzystanie ze środowiska programistycznego takiego jak RStudio lub Visual Studio Code, największą zaletą okazuje się możliwość przekazywania wierszy kodu ze skryptu o rozszerzeniu *.R* do konsoli R w celu ich wykonania. Ta elastyczność zapewnia możliwość stosowania się do najlepszych praktyk związanych z traktowaniem kodu źródłowego jako prawdziwego[3] (w przeciwieństwie do obiektów w przestrzeni roboczej) oraz z zapisywaniem plików *.R* (w przeciwieństwie do zapisywania i wczytywania *.Rdata*).

[3] Cytując odwołanie do filozofii znanej z Emacs Speaks Statistics (ESS), na temat której więcej znajdziesz na stronie *https://ess.r-project.org/Manual/ess.html#Philosophies-for-using-ESS_0028R_0029*.

Funkcja `load_all()` jest ważna podczas tworzenia pakietów i, co można uznać za ironię, wymaga, aby *nie* testować kodu pakietu w taki sam sposób jak testowany jest kod skryptu. Ta funkcja *symuluje* pełny proces w celu poznania efektu wprowadzenia zmiany w kodzie źródłowym, co okazuje się na tyle niewygodne[4], że nie chcesz tego robić zbyt często. Na rysunku 4.4 możesz zobaczyć, że funkcja `library()` wczytuje jedynie pakiet zainstalowany, podczas gdy funkcja `load_all()` oferuje pełną symulację, na podstawie bieżącego kodu źródłowego pakietu.



Rysunek 4.4. Wywołanie devtools::load_all() kontra library()

Do najważniejszych zalet funkcji `load_all()` zaliczamy:

- Możliwość szybkiej iteracji, co z kolei zachęca do eksploracji i stopniowego postępu.
- Przyrostowe przyśpieszenie jest szczególnie widoczne w przypadku pakietów zawierających skompilowany kod.
- Możliwość interaktywnego tworzenia pakietu w ramach reżimu przestrzeni nazw, który poprawnie odzwierciedla sytuację użycia przez kogoś Twojego zainstalowanego pakietu. Oto dodatkowe zalety pojawiające się przy tej okazji:
  - Masz możliwość bezpośredniego wywoływania funkcji wewnętrznych, bez konieczności używania składni `:::` oraz bez tymczasowego definiowania funkcji w globalnej przestrzeni roboczej.
  - Masz możliwość wywoływania funkcji z innych pakietów zaimportowanych do Twojej przestrzeni roboczej, bez konieczności dołączania zależności za pomocą wywołania `library()`.

Funkcja `load_all()` ułatwia pracę podczas tworzenia pakietów i eliminuje pokusę stosowania obejść, których skutkiem często są błędy związane z przestrzenią nazw i zarządzaniem zależnościami.

Inne sposoby wywołania load_all()

Wywołanie `devtools::load_all()` to rodzaj niewielkiego opakowania dla wywołania `pkgload::load_all()`, które okazuje się nieco bardziej przyjazne użytkownikowi. Istnieje niewielkie prawdopodobieństwo, że wywołania `load_all()` będziesz używać w sposób programowy bądź w innym pakiecie. Jeżeli jednak pojawi się taka konieczność, lepiej będzie bezpośrednio użyć `pkgload::load_all()`.

[4] Podejście stosowane w powłoce polega na opuszczeniu języka R, przejściu do powłoki, wydaniu polecenia `R CMD build` w katalogu nadrzędnym pakietu, wydaniu polecenia `R CMD INSTALL foo_x.y.z.tar.gz`, ponownym uruchomieniu R i wywołaniu `library(foo)`.

RStudio

Podczas pracy w Projekcie będącym pakietem środowisko RStudio oferuje kilka sposobów wywołania `load_all()`:

- użycie skrótu klawiszowego *Cmd+Shift+L* (macOS) lub *Ctrl+Shift+L* (Windows i Linux);
- użycie menu *More* w panelu *Build*;
- użycie opcji menu *Build/Load All*.

check() i R CMD check

Bazowe środowisko języka R oferuje dostęp do różnych narzędzi powłoki, a polecenie `R CMD check` to oficjalna metoda sprawdzenia poprawności danego pakietu R. Duże znaczenie ma wykonywanie tego polecenia, jeśli zamierzasz przekazać pakiet do repozytorium CRAN. Mimo to *gorąco zachęcamy* do przestrzegania tego standardu, nawet jeśli nie planujesz przekazania pakietu do repozytorium CRAN. Polecenie `R CMD check` potrafi wykryć wiele najczęściej występujących problemów, których wychwycenie w przeciwnym razie będzie znacznie trudniejsze.

Spójrz na zalecany sposób wywołania `R CMD check` w konsoli za pomocą pakietu devtools:

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

Zalecamy takie rozwiązanie, ponieważ pozwala ono wykonać polecenie `R CMD check` w konsoli R, co z kolei znacznie zmniejsza tarcie i zwiększa prawdopodobieństwo wczesnego i częstego stosowania wywołania `check()`. Ten nacisk na elastyczność i jak najszybsze otrzymanie informacji zwrotnych ma dokładnie tę samą motywację, jaka kryje się za wywołaniem `load_all()`. W przypadku `check()` tak naprawdę mamy do czynienia z wykonywaniem polecenia `R CMD check`. To nie jest jedynie wierna symulacja, jak ma to miejsce w przypadku wywołania `load_all()`.

RStudio

Środowisko RStudio udostępnia wywołanie `check()` w menu *Build*, w panelu *Build*, za pomocą opcji *Check* oraz poprzez skrót klawiszowy *Ctrl+Shift+E* (Windows) lub *Cmd+Shift+E* (macOS).

Często popełniany przez początkujących programistów pakietów R błąd polega na wykonywaniu zbyt wielu operacji w pakiecie pomiędzy poszczególnymi poleceniami `R CMD check`. Gdy to polecenie wreszcie zostanie wykonane, okazuje się, że w pakiecie istnieje wiele problemów, co może być niezwykle demotywujące. Wprawdzie to może wydawać się nieintuicyjne, ale kluczem do zmniejszenia poziomu błędów jest częstsze wykonywanie polecenia `R CMD check`. Im szybciej wychwycisz problem, tym łatwiej go usuniesz[5]. Takie podejście wyjaśniliśmy dość dokładnie w rozdziale 1.

[5] Zapoznaj się z postem *Frequency Reduces Difficulty* opublikowanym przez Martina Fowlera na jego blogu pod adresem *https://martinfowler.com/bliki/FrequencyReducesDifficulty.html*. Autor wyjaśnia w nim podejście typu „jeśli to uciążliwe, rób to znacznie częściej".

Górna granica w takim podejściu polega na wykonywaniu polecenia `R CMD check` po wprowadzeniu każdej zmiany. Podczas aktywnej pracy nad pakietem nie mamy do czynienia z częstym ręcznym wywoływaniem `check()`, przeważnie będzie się to odbywało kilkukrotnie w ciągu dnia. Nie czekaj z wydaniem polecenia `R CMD check` całe dnie, tygodnie lub miesiące aż do osiągnięcia pewnego etapu specjalnego. Jeżeli używasz repozytorium w serwisie GitHub (rozdział 20.), to przedstawimy Ci rozwiązanie, w którym polecenie `R CMD check` będzie wykonywane automatycznie w trakcie każdej operacji przekazania kodu do repozytorium (rozdział 20.).

Sposób pracy

Oto co się dzieje podczas wykonywania wywołania `devtools::check()`:

- Sprawdzenie aktualności dokumentacji za pomocą wywołania `devtools::document()`.
- Utworzenie paczki pakietu przed jego sprawdzeniem (rozdział 3.). Jest to przykład najlepszej praktyki podczas sprawdzania pakietu, ponieważ mamy pewność, że ta operacja będzie przeprowadzona dla pakietu w czystym stanie. Skoro paczka pakietu nie zawiera żadnych plików tymczasowych, które mogły nagromadzić się w katalogu pakietu w postaci kodu źródłowego, np. artefaktów typu pliki *.so* i *.o* towarzyszących skompilowanemu kodowi, to można uniknąć nieprawdziwych ostrzeżeń generowanych podczas przetwarzania takich plików.
- Przypisanie wartości `"true"` zmiennej środowiskowej `NOT_CRAN`. To pozwala na selektywne pomijanie testów w repozytorium CRAN. Zapoznaj się z wynikiem wykonania polecenia `?testthat::skip_on_cran` i informacjami zamieszczonymi w rozdziale 15.

Sposób pracy podczas sprawdzania pakietów jest prosty, ale uciążliwy:

1. Wywołanie `devtools::check()` bądź użycie skrótu klawiszowego *Ctrl/Cmd+Shift+E*.
2. Usunięcie pierwszego problemu.
3. Powtarzanie procedury aż do usunięcia wszystkich problemów.

Wynikiem działania polecenia `R CMD check` jest komunikat jednego z trzech typów:

`ERROR`
: Wskazuje na poważny problem, który należy usunąć niezależnie od tego, czy pakiet będzie przekazany do repozytorium CRAN.

`WARNING`
: Prawdopodobnie wskazuje na problem, który należy usunąć, jeśli planujesz przekazanie pakietu do repozytorium CRAN (jeżeli nie masz takich planów, to i tak warto przyjrzeć się temu problemowi).

`NOTE`
: Niewielki problem lub w pewnych sytuacjach jedynie obserwacja. Jeśli planujesz przekazanie pakietu do repozytorium CRAN, musisz dążyć do wyeliminowania wszystkich komunikatów typu `NOTE`, nawet gdy są to tylko fałszywe alarmy. Brak tego typu komunikatów oznacza, że nie ma konieczności udziału człowieka i proces przekazania pakietu do repozytorium CRAN odbędzie się szybciej. Kiedy nie ma możliwości usunięcia komunikatu typu `NOTE`, trzeba to

wyjaśnić w komentarzu podczas operacji przekazywania pakietu, jak to dokładniej wyjaśniliśmy w rozdziale 22. Jeżeli nie zamierzasz przekazywać pakietu do repozytorium CRAN, dokładnie zapoznaj się z każdym problemem oznaczonym jako `NOTE`. Gdy usunięcie problemu jest proste, warto to zrobić i kontynuować wysiłek w celu otrzymania całkowicie czystego wyniku. Jeżeli jednak wyeliminowanie takiego komunikatu będzie miało negatywny wpływ na pakiet, rozsądne może być tolerowanie komunikatu. Upewnij się, że nie będzie to prowadziło do ignorowania innych problemów, które powinny zostać wyeliminowane.

Polecenie `R CMD check` składa się z dziesiątek różnych operacji sprawdzenia, których wymienienie tutaj byłoby przytłaczające. Więcej informacji na ich temat znajdziesz na stronie *https://r-pkgs.org/R-CMD-check.html*.

Pod maską polecenia R CMD check

Gdy zdobędziesz większe doświadczenie w tworzeniu pakietów, w pewnym momencie zechcesz uzyskać bezpośredni dostęp do polecenia `R CMD check`. Pamiętaj, że to polecenie należy wydać z poziomu powłoki, a nie konsoli R. Zapoznaj się z dokumentacją wyświetlaną po wydaniu następującego polecenia:

```
R CMD check --help
```

Polecenie `R CMD check` może zostać wykonane w katalogu zawierającym pakiet R w postaci kodu źródłowego (rozdział 3.) lub lepiej z poziomu katalogu zawierającego pakiet w postaci paczki (rozdział 3.).

```
R CMD build somepackage
R CMD check somepackage_0.0.0.9000.tar.gz
```

Aby dowiedzieć się więcej na ten temat, zajrzyj się sekcji „Checking packages” książki pt. *Writing R Extensions* (*https://cran.r-project.org/doc/manuals/R-exts.html#Checking-packages*).

Oto doskonały podręcznik tworzenia pakietów dla początkujących i zaawansowanych uzytkowników!

Maëlle Salmon

W języku R podstawową jednostką współdzielonego kodu jest pakiet. Ma on ściśle określoną strukturę i można go łatwo udostępniać innym. Obecnie programiści R mogą korzystać z ponad 19 tysięcy przeróżnych pakietów. Poza prostym pobieraniem i używaniem pakietów opracowanych przez kogoś innego, programistom R przydaje się umiejętność ich samodzielnego tworzenia.

Oto znakomity przewodnik po budowaniu pakietów R. Pokazuje, jak dokładnie wygląda proces tworzenia pakietu i z czego wynika jego struktura. Omawia poszczególne komponenty i metadane pakietu R, wyjaśnia także, na czym polega korzystanie z zależności i jakie są zasady eksportowania funkcji z pakietu. Wyczerpujące wyjaśnienie zagadnień testowania kodu za pomocą pakietu testthat uwzględnia również techniki przydatne w trudniejszych przypadkach. Książka zawiera ponadto omówienie systemu dokumentowania zawartości pakietu, a w końcowych rozdziałach przedstawia praktyki stosowane podczas jego tworzenia, takie jak korzystanie z kontroli wersji i przekazywanie go do repozytorium CRAN.

W książce:

- z czego się składa pakiet R
- praca z pakietem devtools w środowisku RStudio
- tworzenie testów jednostkowych za pomocą pakietu testthat
- przygotowywanie estetycznej i funkcjonalnej dokumentacji przy użyciu pakietu pkgdown
- korzystanie z nowoczesnych platform hostingowych dla kodu źródłowego
- dobre praktyki podczas pracy z pakietami R

Hadley Wickham jest głównym badaczem w firmie Posit, laureatem nagrody COPSS i członkiem R Foundation.

Jennifer Bryan jest inżynierem oprogramowania w firmie Posit, członkinią R Foundation i zespołu tidyverse.

Ułatwisz wielokrotne korzystanie z kodu R – sobie lub innym użytkownikom.

Sam Lau, autor książki *Learning Data Science*

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1046-1
9 788328 910461
Cena: 79,00 zł