



# Tworzenie pakietów R oraz praca z systemem kontroli wersji Git i platformą GitHub

Jakub Nowosad

2024-02-07



# Spis treści

<b>1</b>	<b>Wprowadzenie</b>	<b>6</b>
1.1	Cześć, jestem Jakub . . . . .	6
1.2	Plan szkolenia . . . . .	6
1.3	Format szkolenia . . . . .	6
1.4	Wymagania wstępne . . . . .	8
<b>2</b>	<b>Tworzenie pakietów R (1)</b>	<b>9</b>
2.1	Pakiety R . . . . .	9
2.2	Nazwa pakietu . . . . .	11
2.3	Tworzenie szkieletu pakietu . . . . .	11
2.4	Dokumentacja pakietu . . . . .	12
2.5	Opis pakietu . . . . .	12
2.6	Rozwijanie pakietu . . . . .	13
2.7	Tworzenie i dokumentacja funkcji . . . . .	13
2.8	Zależności . . . . .	15
2.9	Sprawdzanie pakietu . . . . .	15
2.10	Instalowanie pakietu . . . . .	16
<b>3</b>	<b>Kontrola wersji (1)</b>	<b>17</b>
3.1	System Git . . . . .	17
3.2	GitHub . . . . .	18
3.2.1	Tworzenie zdanego repo . . . . .	18
3.2.2	Repozytorium GitHub . . . . .	21
3.2.3	Dodatkowe możliwości GitHub . . . . .	22
3.3	Konfiguracja systemu Git . . . . .	22
3.4	Kontrola wersji w RStudio . . . . .	23
3.5	Sposoby pracy z systemem Git . . . . .	24
3.5.1	Nowy projekt . . . . .	25
3.5.2	Istniejący projekt . . . . .	26
3.6	Problemy z kontrolą wersji . . . . .	26
<b>4</b>	<b>Tworzenie pakietów R (2)</b>	<b>28</b>
4.1	Publikowanie pakietów . . . . .	28
4.2	Metadane pakietu . . . . .	28
4.2.1	Tytuł pakietu . . . . .	29
4.2.2	Wersja pakietu . . . . .	29

4.2.3	Autorzy . . . . .	29
4.2.4	Opis pakietu . . . . .	30
4.2.5	Licencja . . . . .	30
4.2.6	Inne pola . . . . .	31
4.3	Dokumentacja pakietu . . . . .	31
4.3.1	README . . . . .	32
4.3.2	Winiety . . . . .	32
4.3.3	NEWS . . . . .	33
<b>5</b>	<b>Kontrola wersji (2)</b>	<b>34</b>
5.1	Git . . . . .	34
5.2	Repozytorium . . . . .	34
5.3	Dodawanie zmian . . . . .	34
5.4	Sprawdzanie zmian . . . . .	35
5.5	Zatwierdzanie zmian . . . . .	35
5.6	Rozgałęzienia . . . . .	35
5.7	Repozytorium zdalne . . . . .	36
5.8	Wysyłanie zmian . . . . .	37
5.9	Aktualizowanie zmian . . . . .	37
<b>6</b>	<b>Tworzenie pakietów R (3)</b>	<b>38</b>
6.1	Dane w pakietach R . . . . .	38
6.1.1	Dane w postaci binarnej w folderze data/ . . . . .	38
6.1.2	Dane w postaci oryginalnej w podfolderze folderu inst/ . . . . .	39
6.1.3	Dane w postaci binarnej w pliku R/sysdata.rda . . . . .	41
6.1.4	Dane stworzone wewnątrz kodu R . . . . .	42
6.2	Testy jednostkowe . . . . .	42
6.3	Wbudowane testy . . . . .	44
6.4	Pokrycie kodu testami jednostkowymi . . . . .	45
6.5	Strona internetowa pakietu . . . . .	46
6.6	CI/CD . . . . .	46
6.7	CRAN . . . . .	48
<b>7</b>	<b>Kontrola wersji (3)</b>	<b>51</b>
7.1	GitHub issues . . . . .	51
7.2	Pull requests . . . . .	51
7.3	Model pracy . . . . .	52
7.4	Konflikty Git . . . . .	53
7.5	Inne kwestie związane z systemem Git i serwisem GitHub . . . . .	56
7.5.1	Identyfikatory . . . . .	56
7.5.2	Szablony . . . . .	57
7.5.3	Ograniczenia wielkości plików . . . . .	57
7.5.4	GitHub Pages . . . . .	57
7.6	Dodatkowe materiały . . . . .	58

<b>8 Tworzenie pakietów R (4)</b>	<b>60</b>
8.1 Zaawansowane dokumentowanie funkcji . . . . .	60
8.2 Dodatkowe możliwości pakietu <b>usethis</b> . . . . .	62
8.3 Inne kwestie związane z tworzeniem pakietów R . . . . .	63
8.4 Dodatkowe materiały . . . . .	64
<b>Bibliografia</b>	<b>65</b>
<b>Załączniki</b>	<b>66</b>
<b>A Profiling</b>	<b>66</b>
<b>B Benchmarking</b>	<b>69</b>
<b>C C++</b>	<b>72</b>
C.1 Wywoływanie kodu C++ wewnątrz skryptu R . . . . .	73
C.2 Wywoływanie kodu z plików .cpp . . . . .	77
C.3 C++ w pakietach R . . . . .	78

# 1 Wprowadzenie

## 1.1 Cześć, jestem Jakub

Strona internetowa: <https://jakubnowosad.com/>

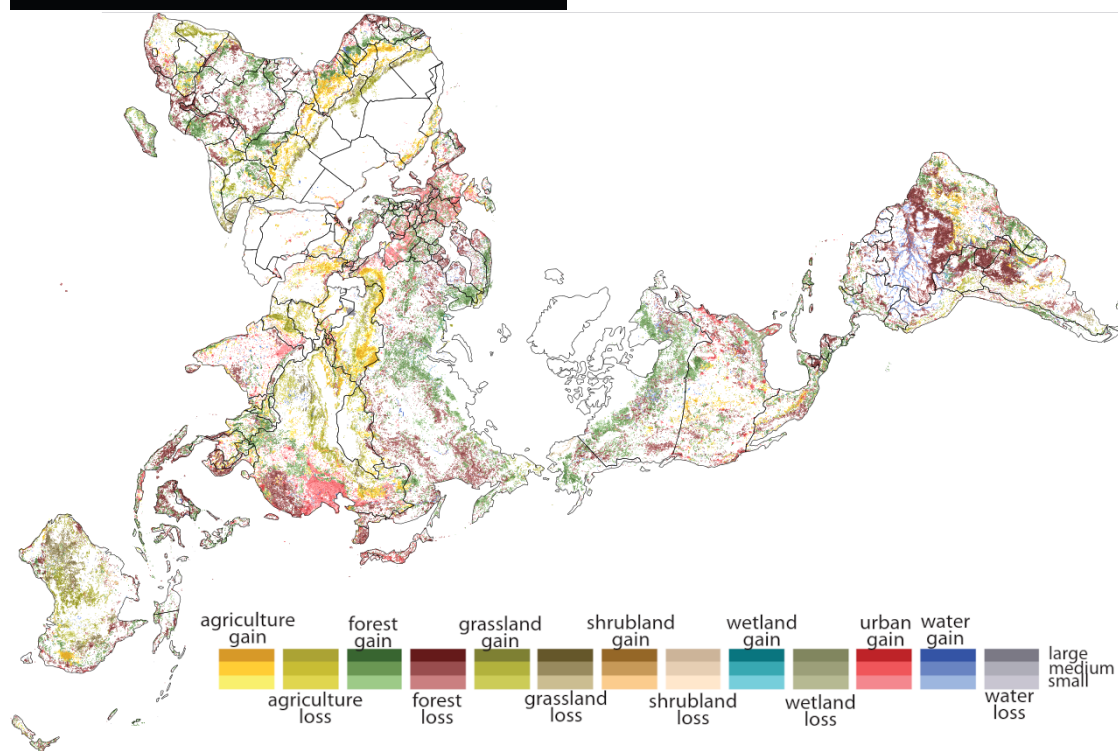
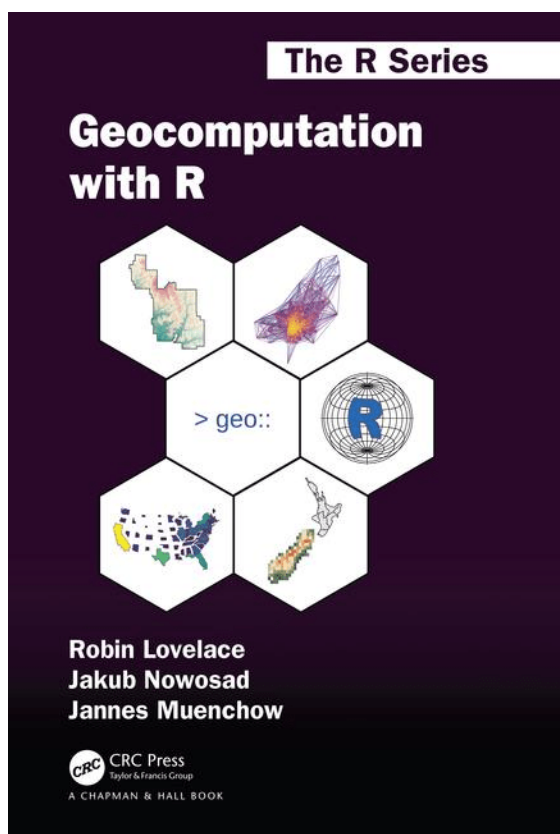
- Geoinformatyk pracujący na styku geografii, informatyki, statystyki i nauk o środowisku
- Adiunkt na Uniwersytecie Adama Mickiewicza w Poznaniu
- Główne zainteresowania naukowe: [analiza struktur przestrzennych, ze szczególnym uwzględnieniem kwantyfikacji i zrozumienia struktur w danych środowiskowych](#)
- Współautor kilku książek i podręczników dotyczących geoinformacji i analizy przestrzennej z wykorzystaniem R i Pythona, w tym [“Geocomputation with R”](#)
- Twórca i współtwórca [różnych pakietów R do przetwarzania i wizualizacji danych przestrzennych](#)
- Edukator, prowadzący kursy z zakresu geoinformatyki, analizy przestrzennej, programowania i wizualizacji danych

## 1.2 Plan szkolenia

Celem szkolenia jest zapoznanie uczestników z możliwościami systemu kontroli wersji Git oraz platformy GitHub, szczególnie w kontekście tworzenia pakietów R. Szkolenie rozpocznie się od przedstawienia podstaw tworzenia pakietów R, a następnie pokaże w jaki sposób umieścić stworzone pakiety na platformie GitHub używając podstawowych komend systemu kontroli wersji Git. Następnie omówione zostaną możliwości i ograniczenia platformy GitHub, ze szczególnym uwzględnieniem specyfiki pracy badawczej. Dalej nastąpi pokazanie średnio-zaawansowanych kwestii związanych z tworzeniem pakietów R (w tym dostosowaniem ich do wymagań repozytorium CRAN), po czym planowane jest skupienie się na aspektach możliwości współpracy wielu osób na platformie GitHub (w tym omówienie często spotykanych problemów). Szkolenie zakończy się wskazaniem zaawansowanych kwestii dotyczących tworzenia pakietów i używania systemów kontroli wersji.

## 1.3 Format szkolenia

Szkolenie będzie składało się z dwóch dni (14h) wykładów, ćwiczeń komputerowych oraz studiów przypadków. Dostarczy również praktycznych wskazówek, jak rozwiązywać



Przykłady moich działań

problemy związane ze współpracą przy użyciu systemu kontroli wersji Git oraz z tworzeniem pakietów. Szkolenie odbędzie się 7-8 lutego 2024 roku.

## 1.4 Wymagania wstępne

Do odtworzenia przykładów oraz do wykonania zadań zawartych w tym dokumencie konieczne jest posiadanie aktualnej wersji **R**. Pod adresem <https://cloud.r-project.org/> można znaleźć instrukcje instalacji R dla systemów Windows, Mac OS i Linux.

W niektórych rozdziałach użyte zostanie zintegrowane środowisko programistyczne **RStudio**. Można je zainstalować korzystając ze strony <https://posit.co/download/rstudio-desktop/#download>.

Aspekty dotyczące kontroli wersji zostaną omówione używając oprogramowania **Git**. Zalecanym sposobem instalacji Git na Windows jest wersja ze strony <https://gitforwindows.org/>. Instrukcja instalacji na system Mac OS znajduje się pod adresem <https://happygitwithr.com/install-git.html#macos>. Wersję Linuxową można zainstalować używając poniższej linii kodu:

```
# Ubuntu  
sudo apt install git
```

```
# Fedora  
sudo dnf install git
```

Należy również posiadać konto na platformie **GitHub**. Warto wtedy zastanowić się nad wyborem nazwy użytkownika, ponieważ jest ona sposobem identyfikacji oraz kontaktu z innymi użytkownikami na GitHubie.<sup>1</sup>

---

<sup>1</sup>Zmiana nazwy jest możliwa później.



## 2 Tworzenie pakietów R (1)

### 2.1 Pakiety R

Pakiet to zorganizowany zbiór funkcji, który rozszerza możliwości R. Pakiety oprócz kodu zawierają szereg dodatkowych istotnych elementów, takich jak:

- Informacja o wersji pakietu, jego twórcach, zależnościach, czy licencji
- Dokumentacja
- Przykładowe dane
- Testy kodu

Pakiety R mogą być przechowywane i instalowane z wielu miejsc w internecie. Istnieje jednak centralne repozytorium (CRAN, ang. *the Comprehensive R Archive Network*), które zawiera oficjalne wersje pakietów R, np. <https://cran.r-project.org/package=stringr>. Wersje deweloperskie (rozwojowe) często można znaleźć na platformie [GitHub](https://github.com/tidyverse/stringr/), np. <https://github.com/tidyverse/stringr/>.

Do instalacji pakietu w R z repozytorium CRAN służy wbudowana funkcja `install.packages()`, np:

```
install.packages("stringr") #instalacja pakietu stringr
```

Zainstalowanie pakietu w R z platformy GitHub jest możliwe używając, np. funkcji `install_github()` z pakietu **remotes**.

```
# install.packages("remotes")
remotes::install_github("tidyverse/stringr")
```

W przypadku instalacji pakietu w R z platformy GitHub należy podać nazwę użytkownika lub organizacji, która tworzy ten pakiet (np. powyżej `tidyverse`) oraz nazwę pakietu (np. powyżej `stringr`) oddzielone znakiem `/`.

Podobnie jak instalowanie programów na komputerze - zainstalowanie pakietu odbywa się tylko jeden raz.

#### Komentarz

Istnieją dwa główne formy, w których rozpowszechniane są pakiety R - postać źródłowa (ang. *source packages*) i postać binarna (ang. *binary packages*). Postać źródłowa zawiera kod źródłowy pakietu, który musi zostać następnie skompilowany na komputerze użytkownika. Skompilowanie pakietu na podstawie kodu źródłowego może wymagać posiadania odpowiednich bibliotek na komputerze, np. [Rtools](#) dla systemu Windows. Dodatkowo, instalacja w ten sposób zabiera więcej czasu. Postać binarna została już wcześniej skompilowana na zewnętrznym komputerze (np. w repozytorium CRAN) Jest ona dostępna dla systemów Windows i Mac OS. Niestety, nie wszystkie pakiety (lub ich wersje) posiadają postać binarną i wymagana jest ich kompilacja.

Użycie wybranego pakietu wymaga dołączenia go do R za pomocą funkcji `library()`. Dołączenie wybranych pakietów do R robimy po każdym uruchomieniu R.

```
library(stringr)
```

#### Komentarz

Pakiet (ang. **package**) to zbiór funkcji, biblioteka (ang. **library**) to miejsce na dysku, w którym znajdują się pakiety.

W przypadku, gdy chcemy użyć zewnętrznej funkcji, ale nie dołączyliśmy odpowiedniego pakietu, pojawi się błąd o treści `could not find function "nazwa_funkcji"`.

```
str_sub("chronologia", start = 1, end = 6)
#> Error in str_sub("chronologia", start = 1, end = 6) :
#> could not find function "str_sub"
```

Istnieją dwa możliwe rozwiązania powyższego problemu. Po pierwsze możliwe jest dołączenie pakietu poprzez `library(stringr)`. Po drugie można bezpośrednio zdefiniować z jakiego pakietu pochodzi konkretna funkcja używając nazwy pakietu i operatora `::`.

```
stringr::str_sub("chronologia", start = 1, end = 6)
```

```
[1] "chrono"
```

#### Komentarz

Operator `::` może być też pomocny w przypadku, gdy kilka pakietów ma funkcję o tej samej nazwie. Wówczas, aby kod został poprawnie wykonany, warto podać nie tylko nazwę funkcji ale też nazwę pakietu z jakiego ona pochodzi.

## 2.2 Nazwa pakietu

Nazwa nowego pakietu musi spełniać kilka wymagań: składać się tylko ze znaków [ASCII](#), cyfr i kropek, mieć co najmniej dwa znaki oraz zaczynać się od litery i nie kończyć się kropką (R Core Team 2019). Ważne jest również myślenie o nazwie pakietu tak jak o nazwach funkcji – nazwy pakietów powinny ułatwiać zrozumienie ich zawartości. Dodatkowo, z uwagi na istnienie wielu pakietów, warto najpierw sprawdzić czy pakiet o wymyślonej przez nas nazwie już nie istnieje. Można to przykładowo zrobić używając pakietu **available** (Ganz i in. 2019), który sprawdza przy wybrana nazwa nie jest już zajęta oraz czy nie ma ona jakiegoś niepożądanego przez nas znaczenia.

```
# install.packages("available")
available::available("mojpakiet", browse = FALSE)
```

```
'getOption("repos")' replaces Bioconductor standard repositories, see
'help("repositories", package = "BiocManager")' for details.
```

Replacement repositories:

CRAN: <https://cloud.r-project.org/>

```
-- mojpakiet -----
Name valid: [?]
Available on CRAN: [?]
Available on Bioconductor: [?]
Available on GitHub: [?]
Bad Words: mojpakiet
Abbreviations: http://www.abbreviations.com/mojpakiet
Wikipedia: https://en.wikipedia.org/wiki/mojpakiet
Wiktoryary: https://en.wiktoryary.org/wiki/mojpakiet
Sentiment:???
```

## 2.3 Tworzenie szkieletu pakietu

Kolejnym krokiem jest stworzenie szkieletu pakietu, czyli zorganizowanego zbioru plików i folderów, do których później należy dodać odpowiednie informacje i funkcje. Znacznie w tym może pomóc pakiet **usethis** (Wickham i Bryan 2020), który zawiera szereg funkcji ułatwiających budowanie pakietów R.

```
library(usethis)
```

Do stworzenia szkieletu pakietu służy funkcja `create_packages()`, w której należy podać ścieżkę do nowego pakietu. W tej ścieżce ostatnia nazwa folderu określa również nazwę pakietu.<sup>1</sup>

```
usethis::create_package("~/Documents/mojpakiet")
```

W efekcie działania powyższej funkcji stworzony zostanie nowy folder `mojpakiet` zawierający kilka plików oraz otwarty zostanie nowy projekt RStudio zawierający ten pakiet. Najważniejsze nowe pliki to:

1. `mojpakiet.Rproj` - plik projektu RStudio
2. `DESCRIPTION` - plik zawierający podstawowe informacje o pakiecie
3. `R/` - w tym pustym folderze konieczne będzie umieszczenie nowych funkcji R
4. `NAMESPACE` - ten plik określa, między innymi, jakie funkcje są dostępne w tym pakiecie. Ten plik i jego zawartość jest tworzona automatycznie

Dodatkowo w prawym górnym panelu RStudio pojawi się nowy panel “Build”.

## 2.4 Dokumentacja pakietu

Dokumentowanie pakietu ma miejsce na wielu poziomach, począwszy od opisu pakietu w pliku `DESCRIPTION`, poprzez komentowanie kodu, dokumentację funkcji wraz z przykładami jej użycia, dokumentację danych, plik `README`, winiety, aż po stronę internetową pakietu.

## 2.5 Opis pakietu

Plik `DESCRIPTION` zawiera opis (metadane) pakietu, w tym jego nazwę, tytuł, wersję, autorów, opis, czy licencję.

```
Package: mojpakiet
Title: Moje Funkcje Robiace Wszystko
Version: 0.0.1
Authors@R:
  person(given = "Imie",
         family = "Nazwisko",
         role = c("cre", "aut"),
         email = "imie.nazwisko@example.com")
Description: Tworzenie, przeliczanie i wyliczanie wszystkiego.
  Czasami nawet więcej.
License: CC0
```

---

<sup>1</sup>Funkcja również `create_packages()` sama tworzy nowy folder, jeżeli on wcześniej nie istniał.

```
Encoding: UTF-8
LazyData: true
RoxygenNote: 7.2.3
```

Plik DESCRIPTION należy regularnie uaktualniać, np. zmieniać numer wersji po naniesionych zmianach w kodzie, czy dodawać nowych autorów, jeżeli tacy się pojawili.

Więcej o tym później (Sekcja 4.2).

## 2.6 Rozwijanie pakietu

Rozwój pakietu R może opierać się na kilku poniższych krokach:

1. Tworzenie/modyfikowanie kodu w folderze R/
2. Używanie funkcji `devtools::load_all()`, która dodaje nowe/zmodyfikowane funkcje do środowiska R
3. Sprawdzenie czy funkcja działa zgodnie z oczekiwaniami na kilku przykładach
4. Dodanie testów jednostkowych na podstawie stworzonych przykładów (o tym więcej w Sekcja 6.2)
5. Uaktualnienie dokumentacji tworzonego/modyfikowanego kodu
6. Wygenerowanie plików z dokumentacją używając `devtools::document()`
7. Sprawdzenie czy pakiet nie posiada żadnych problemów używając `devtools::test()` oraz `devtools::check()`
8. Modyfikacja wersji oprogramowania w pliku DESCRIPTION
9. Zapisanie zmian w kodzie w repozytorium (np. GitHub)
10. Powtórzenie powyższych czynności przy kolejnych zmianach w kodzie

### **i** ZADANIA

1. Stwórz nowy pakiet R o wybranej przez siebie nazwie: zacznij od sprawdzenia czy nazwa pakietu nie jest już zajęta.
2. Stwórz szkielet pakietu oraz uzupełnij najważniejsze informacje w pliku DESCRIPTION.

## 2.7 Tworzenie i dokumentacja funkcji

```
konwersja_temp = function(temperatura_f){
  (temperatura_f - 32) / 1.8
}
```

Umieszczenie tej funkcji w nowym pakiecie R odbywa się poprzez zapisanie tego kodu jako skrypt R (np. `konwersja_temp.R`) w folderze `R/`.

Funkcje zawarte w pakietach muszą także posiadać odpowiednią dokumentację, zawierającą, między innymi, tytuł funkcji, opis jej działania, wyjaśnienie kolejnych argumentów funkcji, oraz przykłady jej działania. Linie obejmujące dokumentację funkcji rozpoczynają się od znaków `#`, a tworzenie dokumentacji funkcji odbywa się poprzez wypełnianie treści dla kolejnych znaczników (np. `@example` określa występowanie przykładu).

Przykładowy plik `R/konwersja_temp.R` może wyglądać następująco:

```
#' Konwersja temperatur
#'  
#' @description Funkcja sluzaca do konwersji temperatury
#'   ze stopni Fahrenheita do stopni Celsjusza.
#'  
#' @param temperatura_f wektor zawierajacy wartosci temperatury
#'   w stopniach Fahrenheita
#'  
#' @return wektor numeryczny
#' @export
#'  
#' @examples
#' konwersja_temp(75)
#' konwersja_temp(110)
#' konwersja_temp(0)
#' konwersja_temp(c(0, 75, 110))
konwersja_temp = function(temperatura_f){
  (temperatura_f - 32) / 1.8
}
```

Pierwsza linia w tym pliku określa tytuł danej funkcji. Kolejny element rozpoczynający się od znacznika `@description` zawiera krótki opis tego, co funkcja robi. Następnie zazwyczaj wypisane są wszystkie argumenty danej funkcji używając kolejnych znaczników `@param`. Znacznik `@return` pozwala na przekazanie informacji o tym co jest zwracane jako efekt działania funkcji. Przedostatnim znacznikiem w powyższym przypadku jest `@export`. Oznacza on, że ta funkcja będzie widoczna dla każdego użytkownika tego pakietu po użyciu `library(mojpakiet)`. Bez tego znacznika funkcja byłaby tylko widoczna wewnątrz pakietu. Ostatni znacznik, `@examples`, wypisuje kolejne przykłady działania funkcji.

#### Komentarz

Powyższy przykład nie wykorzystuje wszystkich możliwych znaczników. Więcej z nich można znaleźć w dyskusji na stronie <https://github.com/r-lib/roxygen2/issues/792#>

```
issuecomment-705071228.
```

Wybór More -> Document w panelu "Build" (inaczej wywołanie funkcji `devtools::document()` lub użycie skrótu `CTRL+SHIFT+D`) spowoduje zbudowanie pliku dokumentacji w folderze `man`, np. `man/konwersja_temp.Rd`. Pliki dokumentacji będą zawsze tworzone w ten sposób - nie należy ich modyfikować ręcznie. Zbudowanie pliku dokumentacji pozwala teraz na jej podejrzenie poprzez wywołanie pliku pomocy naszej funkcji:

```
?konwersja_temp
```

## 2.8 Zależności

Istnieje jedna ważna różnica pomiędzy tworzeniem funkcji w skryptach a tworzeniem jej wewnątrz pakietu - w pakietach nie można używać dołączania pakietów za pomocą funkcji `library()`. Zamiast tego możliwe jest definiowanie każdej zewnętrznej funkcji używając operatora `::`.<sup>2</sup>

Dodatkowo każda zależność z zewnętrznym pakietem musi być określona w pliku `DESCRIPTION`. Jest to możliwe używając wpisów `Imports:` oraz `Suggests:`, przykładowo:<sup>3</sup>

```
Imports:
  stringr,
  readr
Suggests:
  readxl
```

`Imports:` określa pakiety, które muszą być zainstalowane, aby tworzony pakiet mógł zadziałać. Jeżeli wymienione tutaj pakiety nie będą znajdować się na komputerze użytkownika to zostaną one automatycznie doinstalowane podczas instalacji naszego pakietu. `Suggests:` wymienia pakiety, które pomagają w użytkowaniu naszego pakietu, np. takie które zawierają testowe dane. Wymienione tutaj pakiety nie będą automatycznie doinstalowane podczas instalacji naszego pakietu.

## 2.9 Sprawdzanie pakietu

W momencie, gdy pakiet posiada już swoje podstawowe elementy, tj. pierwsze udokumentowane funkcje oraz uzupełniony opis wraz z zależnościami warto sprawdzić czy te wszystkie elementy pakietu dobrze współgrają ze sobą. Można to zrobić używając funkcji `devtools::check()` (inaczej wybór Check w panelu "Build" RStudio lub skrót `CTRL+SHIFT+E`). W efekcie tego

<sup>2</sup>Istnieją również inne możliwości, np. użycie znaczników `@import` lub `@importFrom`.

<sup>3</sup>Istnieją również inne wpisy, takie jak `Depends:`, `LinkingTo:`, czy `Enhances:`.

wywołania zostanie uruchomiony szereg sprawdzeń i testów dotyczących pakietu, jego funkcji czy opisu. Na końcu zwrócone zostanie wypisanie liczby błędów (*error*), ostrzeżeń (*warnings*) i notatek (*notes*), poprzedzone wymienieniem każdego ich wystąpienia. Błędy oznaczają, że z jakiegoś powodu pakietu nie można zbudować, ostrzeżenia natomiast sugerują sytuację w której jakieś ważne elementy funkcji mogą wymagać poprawy. Notatki natomiast wskazują na kwestie, które użytkownik może, ale nie musi poprawić.

## **i** ZADANIA

1. Dodaj do swojego pakietu twoją funkcję jako plik o rozszerzeniu `.R` w folderze `R\`.
2. Dodaj dokumentację do swojej funkcji, w tym jej tytuł, opis, argumenty, zwracane wartości i przykłady jej użycia.
3. Wygeneruj dokumentację dla swojego pakietu.
4. Sprawdź czy twój pakiet nie posiada żadnych błędów, ostrzeżeń i notatek. W przypadku, gdy masz jakieś błędy lub ostrzeżenia, spróbuj je naprawić.

*Po wykonaniu zadań będziemy mieć czas na dyskusję i pomoc w rozwiązywaniu problemów.*

## 2.10 Instalowanie pakietu

Sprawdzony pakiet, który nie zwraca błędów można zainstalować na własnym komputerze używając funkcji `devtools::install()` (inaczej wybór `Install and restart` w panelu “Build” RStudio lub skrót `CTRL+SHIFT+B`). W przypadku, gdy kod źródłowy tego pakietu znajduje się na platformie GitHub, inni użytkownicy mogą go zainstalować za pomocą funkcji `remotes::install_github("nazwa_uzytkownika_github/nazwa_pakietu")` (Hester i in. 2020).



## 3 Kontrola wersji (1)

Systemy kontroli wersji to narzędzia pozwalające na zapamiętywanie zmian zachodzących w plikach. Dzięki nim możemy sprawdzić nie tylko kiedy zmieniliśmy dany plik i kto go zmienił, ale co najważniejsze - możemy linia po linii prześledzić zmiany wewnątrz tego pliku. Dodatkowo, mamy możliwość przywracania wersji pliku z wybranego czasu w całej historii jego zmian.

Systemy kontroli wersji są bardzo powszechnie wykorzystywane przy tworzeniu wszelakiego rodzaju oprogramowania. Wynika to nie tylko z ich zalet wymienionych powyżej, ale również rozbudowanych możliwości pozwalających na zorganizowaną współpracę wielu osób nad jednym projektem.

Istnieje wiele systemów kontroli wersji różniących się zarówno używaną terminologią, sposobem działania czy możliwościami.<sup>1</sup> Współcześnie najbardziej popularnym systemem kontroli jest Git, a inne popularne systemy kontroli wersji to Concurrent Versions System (CVS), Mercurial czy Subversion (SVN).

### 3.1 System Git

System Git jest niezależny od języka (lub języków) programowania, które używamy. Jego działanie oparte jest o system komend rozpoczynających się od słowa git, które należy wykonać w systemowym oknie konsoli.<sup>2</sup> Zrozumienie działania systemu Git wymaga także poznania kilku nowych terminów.

System Git został zaprojektowany i jest używany głównie do kontroli wersji plików tekstowych. Dzięki temu możemy w prosty sposób zobaczyć, co do linii kodu, w którym miejscu zaszła zmiana. Dodatkowo przechowywanie plików tekstowych i ich zmian nie zajmuje dużo miejsca. Możliwe w systemie Git jest również przechowywanie kolejnych wersji plików binarnych (np. pliki dokumentów, arkusze kalkulacyjne, obrazki, itd.). W ich przypadku niestety nie można liczyć na dokładne sprawdzanie miejsc zmian, a także ich wielkość może powodować znaczne powiększanie się repozytorium.<sup>3</sup>

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Comparison\\_of\\_version-control\\_software#History\\_and\\_adoption](https://en.wikipedia.org/wiki/Comparison_of_version-control_software#History_and_adoption)

<sup>2</sup>Nie w oknie konsoli R.

<sup>3</sup>Między innymi z tego powodu internetowe serwisy kontroli wersji posiadają ograniczenia dotyczące wielkości plików. Przykładowo, GitHub ogranicza wielkość pojedynczych plików do 100MB.

## 3.2 GitHub

GitHub jest serwisem internetowym pozwalającym na przechowywanie i interakcję z repozytoriami w systemie kontroli wersji Git. Posiada on dwa rodzaje repozytoriów - publiczne (ang. *public*), które może każdy zobaczyć oraz prywatne (ang. *private*) dostępne tylko dla osób z odpowiednimi uprawnieniami.

Repozytoria połączone są z kontami użytkowników (np. <https://github.com/Nowosad> to moje konto, gdzie "Nowosad" oznacza nazwę użytkownika) lub organizacjami (np. <https://github.com/r-spatialecology> to konto organizacji "r-spatialecology"). Pod adresem <https://github.com/join> można założyć nowe konto użytkownika.

GitHub może być używany do wielu celów, między innymi:

- Przechowywanie kodu źródłowego
- Tworzenie kopii zapasowych różnych wersji oprogramowania i innych plików tekstowych
- Dzielenie się kodem z innymi
- Współpraca nad kodem
- Hosting statycznych stron internetowych i innych dokumentów
- Hosting pakietów R
- Odkrywanie i wykorzystywanie istniejących projektów
- Śledzenie zmian w kodzie innych osób

Możliwe jest również łączenie możliwości serwisu GitHub z innymi serwisami internetowymi, takimi jak [Codecov](#), [Gitter](#) i [wiele innych](#).

### 3.2.1 Tworzenie zdanego repo

Posiadanie konta użytkownika pozwala na, między innymi, tworzenie nowych repozytoriów i zarządzanie nimi. Stworzenie nowego repozytorium odbywa się poprzez naciśnięcie zielonej ikony.

**Repositories**



Rysunek 3.1: Ikona tworzenia nowego repozytorium GitHub.

W kolejnym oknie należy podać nazwę nowego repozytorium oraz wybrać czy będzie ono publiczne czy prywatne. Dodatkowo możliwe jest dodanie opisu repozytorium (ang. *description*), pliku README, czy licencji.

Po wybraniu potwierdzenia (*Create repository*) utworzone zostanie nowe, puste repozytorium.

## Create a new repository

A repository contains all project files, including the revision history.

---

Owner

Repository name \*

Space A  
spacea ▾


/

Great repository names are short and memorable. Need inspiration? How about **fluffy-invention**?


Description (optional)

---

☐

 **Public**  
Anyone can see this repository. You choose who can commit.


☒

 **Private**  
You choose who can see and commit to this repository.

---

☐ **Initialize this repository with a README**  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

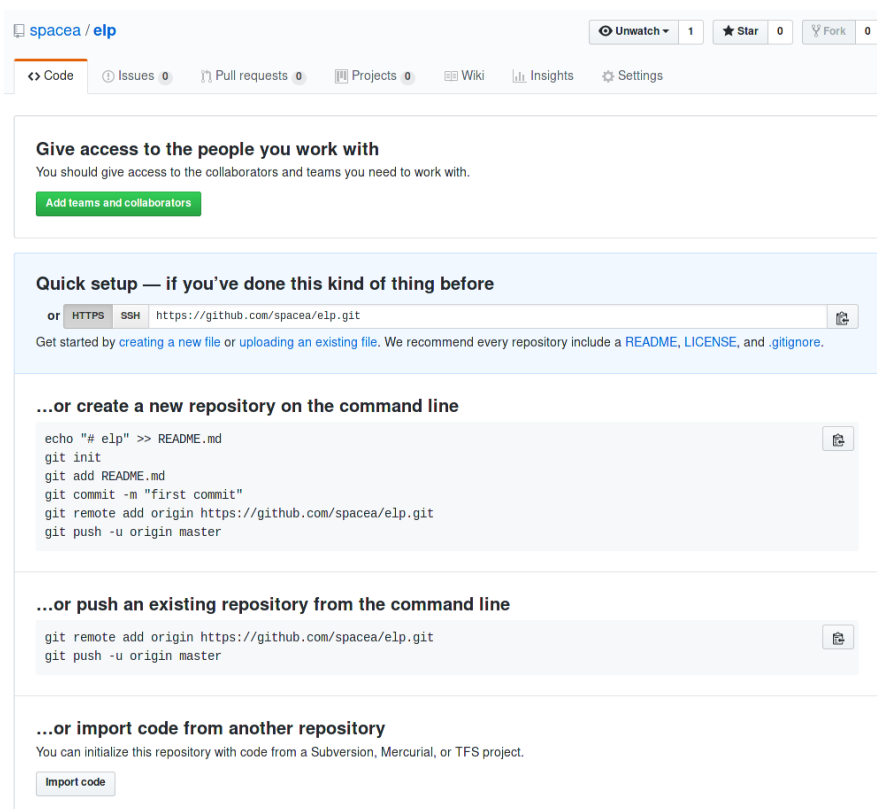
Add .gitignore: **None** ▾

Add a license: **None** ▾ 

---

Create repository

Rysunek 3.2: Okno tworzenia nowego repozytorium GitHub.



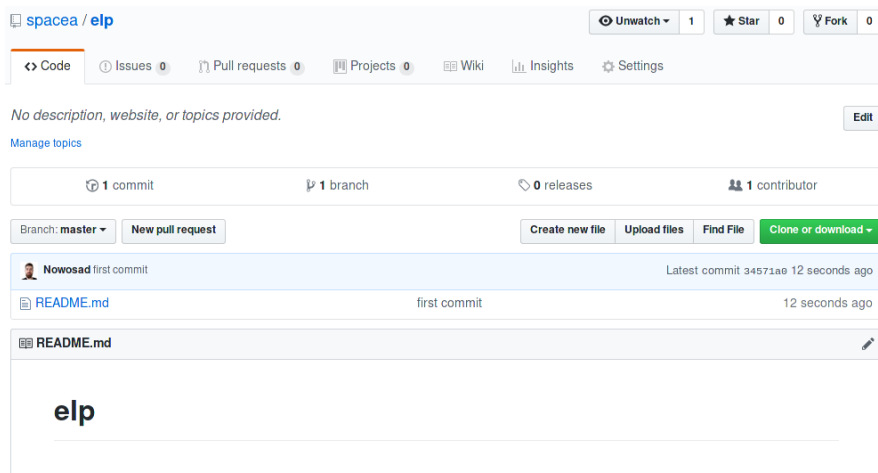
Rysunek 3.3: Nowe, puste repozytorium GitHub.

Okno pustego repozytorium przedstawia cztery główne drogi pozwalające na dodanie zawartości:

1. Szybka konfiguracja - tutaj podane są dwie możliwe ścieżki do zdalnego repozytorium. Pierwsza z nich to adres HTTPS a druga to adres SSH. W sekcji zostanie wyjaśnione jak korzystać z szybkiej konfiguracji.
2. Stworzenie nowego repozytorium używając linii komend. Jest to używane w sytuacjach, gdy lokalna wersja repozytorium jeszcze nie istnieje. W tej sytuacji (1) tworzony jest nowy plik tekstowy README.md, (2) obecny katalog jest określany jako repozytorium Git, (3) plik README.md jest dodawany do repozytorium, (4) dodanie tego pliku jest zatwierdzone wraz z wiadomością "first commit", (5) dodana jest ścieżka do zdalnego repozytorium, (6) następuje wysłanie zmian z lokalnego do zdalnego repozytorium.
3. Wysłanie zmian z istniejącego repozytorium. Ta opcja przydaje się, gdy mamy już istniejące lokalne repozytorium, ale do którego nie ma jeszcze zdalnego repozytorium. Tutaj następuje tylko (1) dodanie ścieżki do zdalnego repozytorium oraz (2) wysłanie zmian z lokalnego do zdalnego repozytorium.
4. Import kodu z innego systemu kontroli wersji niż Git.

### 3.2.2 Repozytorium GitHub

Wygląd okna repozytorium zmienia się po dodaniu pierwszej zawartości.



Rysunek 3.4: Repozytorium GitHub po dodaniu zawartości.

Teraz możliwe jest podejrzanie występujących tam plików (w tym momencie jedynie plik README.md), zmian jakie zaszły w repozytorium (klikając na *commit*), istniejących rozgałęzień (klikając na *branch*) oraz wiele innych. Pod zieloną ikoną *Clone or download* można dodatkowo znaleźć ścieżkę do tego zdalnego repozytorium.

### 3.2.3 Dodatkowe możliwości GitHub

W prawym górnym rogu okna repozytorium znajdują się trzy ikony - *Watch*, *Star*, *Fork*. Pierwsza z nich pozwala na określenie czy chcemy dostawać powiadomienia na temat dyskusji prowadzonych wewnątrz danego repozytorium, takich jak utworzenie nowej sprawy. Druga ikona pozwala na oznaczanie interesujących repozytoriów i przez to ułatwiająca znajdowania podobnych projektów. Ostatnia ikona *Fork* oznacza w tym kontekście rozwidlenie. Po jej kliknięciu następuje utworzenie kopii repozytorium innego użytkownika do naszego konta.

Oprócz dostępu do kodu i jego zmian, GitHub oferuje także szereg dodatkowych możliwości. Obejmuje to, między innymi, automatyczne wyświetlanie plików README, śledzenie spraw (ang. *issue tracking*), zapytania aktualizacyjne (ang. *pull request*), wizualizacje zmian, czy nawet tworzenie stron internetowych. Więcej o tych możliwościach dowiedzie się później.

#### ZADANIA

1. Zaloguj się platformy GitHub. Przejrzyj jej interfejs – czy rozumiesz co tam się znajduje? W razie wątpliwości – pytaj.
2. Załóż nowe repozytorium na GitHubie o dowolnej nazwie.
3. Z poziomu platformy GitHub dodaj nowy plik README.md do repozytorium, w którym podasz nazwę repozytorium oraz swoje imię i nazwisko.
4. Pobierz wersję .zip repozytorium i rozpakuj ją na swoim komputerze.

## 3.3 Konfiguracja systemu Git

Kolejnym krokiem po instalacji systemu Git<sup>4</sup> jest jego konfiguracja. Można ją wykonać używając wbudowanego terminala (Mac OS i Linux) lub terminala dodanego podczas instalacji systemu Git (Windows). Polega ona na podaniu nazwy użytkownika (np. "Imie Nazwisko") oraz jego adresu email ("email@portal.com").

```
git config --global user.name "imie nazwisko"
git config --global user.email "email"
```

Podany adres email powinien być zgodny z tym, który został użyty podczas rejestracji na serwisie GitHub.

Gdy już posiadamy konto na GitHubie oraz repozytorium, przychodzi czas na połączenie go z naszym komputerem. Musimy do tego celu stworzyć a następnie dodać do naszego komputera tzw. GitHub Token. Ma to miejsce na stronie <https://github.com/settings/tokens>, gdzie należy:

---

<sup>4</sup>Instrukcje dotyczące instalacji Gita znajdują się we wstępie książki.

1. Kliknąć przycisk *Generate new token*.
2. Nadać nazwę tokenowi.<sup>5</sup>
3. Zaznaczyć opcje. [Rekomendowane](#) to “repo”, “user” i “workflow”.
4. Kliknąć przycisk *Generate token*.

W efekcie zostanie wygenerowany token, który należy skopiować i zapisać w bezpiecznym miejscu.<sup>6</sup>

Możliwy jest system pracy, w którym podajemy token za każdym razem, gdy chcemy się połączyć z repozytorium GitHub. Dużo jednak wygodniejszym rozwiązaniem jest dodanie tokena na lokalnym komputerze. W tym celu najlepiej użyć funkcji `gitcreds::gitcreds_set()`.

#### Komentarz

Alternatywnym sposobem połączenia repozytorium z komputerem jest wykorzystanie kluczy SSH. W tym celu należy wygenerować klucze SSH, a następnie dodać klucz publiczny do konta na GitHubie. Więcej informacji można znaleźć na stronie <https://help.github.com/en/github/authenticating-to-github/connecting-to-github-with-ssh>.

## 3.4 Kontrola wersji w RStudio

RStudio posiada wbudowane, uproszczone graficzne wsparcie dla systemu Git. Istnieje też szereg programów, których głównym celem jest ułatwienie pracy z systemem Git. Nazwane są one klientami Git, wśród których można wymienić [GitKraken](#) i [Sourcetree](#).<sup>7</sup>

Najprostszym sposobem połączenia RStudio z systemem Git i serwisem GitHub jest stworzenie nowego projektu:

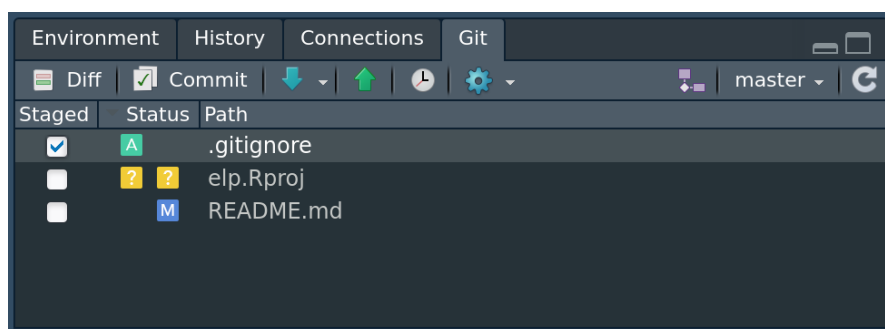
1. Kliknąć *File -> New Project*.
2. Wybrać *Version Control*.
3. Wybrać *Git*.
4. Podać ścieżkę do zdalnego repozytorium (adres HTTPS lub SSH, w zależności od sposobu wybranego wcześniej) oraz wybrać miejsce na dysku, gdzie ma się ten projekt znajdować.
5. Kliknąć *Create Project*.

W efekcie zostanie utworzony nowy projekt RStudio (w tle wykonywane jest pobranie kopii istniejącego zdalnego repo), który jednocześnie jest lokalnym repozytorium Git. Dodatkowo, w RStudio pojawi się nowy panel “Git”.

<sup>5</sup>U mnie to często nazwa komputera na którym pracuję.

<sup>6</sup>Można też użyć funkcji `usethis::create_github_token()`.

<sup>7</sup>Więcej klientów Git można znaleźć na stronie <https://git-scm.com/downloads/guis>.



Rysunek 3.5: Panel Git w RStudio.

W tym panelu są wyświetlone (1) wszystkie pliki, które są w folderze projektu, ale nie w repozytorium Git (żółte ikony statusu), (2) pliki, które chcemy dodać do repozytorium (zielona ikona statusu), oraz (3) pliki, które są już w repozytorium, ale zostały zmodyfikowane (niebieska ikona statusu).<sup>8</sup> Ten panel nie pokazuje plików, które nie zostały ostatnio zmienione. Pierwsza kolumna w tym panelu (*Staged*) domyślnie zawiera same nieodhaczone białe pola. Wybór tego pola (jego odhaczenie) jest równoznaczne z dodaniem zmian.

Dodatkowo nad listą plików znajduje się szereg ikon. Pierwsze dwie z nich (*Diff* i *Commit*) wyświetlają okno, które pozwala sprawdzić jakie zmiany zaszły w plikach od ostatniego ich dodania (dolny panel) oraz zatwierdzić zmiany (prawy panel). Kolejne, strzałki w dół i górę, oznaczają odpowiednio aktualizowanie zmian i wysyłanie zmian. Ikona zegarka otwiera nowe okno, w którym można zobaczyć jakie zmiany zaszły w kolejnych zatwierdzeniach zmian (tak zwanych *commitach*). Następne ikony pozwalają na określenie plików do ignorowania (ikona koła zębatego) oraz tworzenie nowych rozgałęzień. Przedostatni element tego okna to nazwa obecnie ustawionego rozgałęzienia, a po kliknięciu tej nazwy możliwa jest przejście do innego rozgałęzienia.

### 3.5 Sposoby pracy z systemem Git

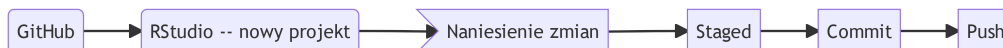
Istnieje wiele możliwych sposobów pracy z systemem Git. Zależą one od wielu czynników, takich jak planowany cel repozytorium czy wykorzystywana technologia. Dodatkowo znaczny wpływ na sposób pracy z systemem Git ma czynnik ludzki - przyzwyczajenia osób pracujących nad projektem i ich preferencje. Poniżej pokażę dwa podstawowe sposoby pracy, a bardziej zaawansowane podejścia zostaną omówione w kolejnych rozdziałach.

<sup>8</sup>Możliwe są też inne sytuacje, np. czerwona ikona z literą R sugerująca zmianę nazwy pliku.



### 3.5.1 Nowy projekt

Preferowanym<sup>9</sup> sposobem rozpoczęcia pracy nad nowym zadaniem (projektem) w R jest stworzenie nowego, pustego repozytorium w serwisie GitHub, a następnie połączenie z nim nowego projektu RStudio.



W momencie, gdy posiadamy ustawione zarówno lokalne jak i zdalne repozytorium możliwe jest rozpoczęcie pracy. Teraz można tworzyć nowe oraz edytować istniejące pliki. Tutaj zalecane jest najpierw kliknięcie ikony aktualizowania zmian (strzałka w dół), aby upewnić się, że posiadamy aktualną wersję repozytorium. Po każdej wyraźnej zmianie plików (np. ulepszenie kodu, naprawa błędów, dodanie nowych możliwości) należy dodać zmiany oraz je zatwierdzić. Można to zrobić klikając pole *Staged* przy wybranych plikach oraz następnie ikonę *Commit*. Teraz można dodać wiadomość opisującą zmiany jakie zaszły, oraz ją zatwierdzić klikając przycisk *Commit*.

Efektem powyższej operacji jest posiadanie zatwierdzonych zmian w lokalnym repozytorium, ale jeszcze ich brak w repozytorium zdalnym. Kolejnym krokiem jest przesłanie zmian na zdalne repozytorium poprzez kliknięcie ikony wysyłania zmian (strzałka w górę). Jeżeli wszystko poszło zgodnie z planem, nowa wersja repozytorium powinna pojawić się na odpowiedniej stronie serwisu GitHub. Tą czynność warto wykonywać rzadziej niż poprzednią, ale też regularnie.

#### Komentarz

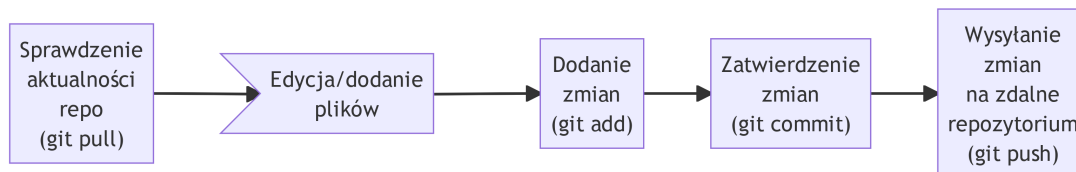
Często w folderze projektu możesz posiadać pliki, których nie chcesz dodawać do repozytorium. W takiej sytuacji dodaj ich nazwy do pliku `.gitignore` i staną się one niewidoczne dla systemu Git.

Dalej praca polega na powtarzaniu tych czynności:

1. Sprawdzenie czy posiadamy aktualną wersję repozytorium.
2. Edycja/dodanie plików czy folderów.
3. Dodanie zmian.
4. Zatwierdzenie zmian.
5. Wysyłanie zmian na zdalne repozytorium.

---

<sup>9</sup>Przeze mnie



### 3.5.2 Istniejący projekt

Czasami posiadasz już jakiś istniejący projekt (folder z kodem źródłowym, itd.), ale chcesz do niego dodać możliwości kontroli wersji. W takich przypadkach najprostszy sposób to stworzenie nowego repozytorium w serwisie GitHub oraz pustego, połączonego z nim nowego projektu RStudio. Następnie należy przekopiować do tego projektu wszystkie już istniejące pliki, dodać je (pole *Staged*), zatwierdzić oraz przesłać na zdalne repozytorium. Kolejne etapy pracy wyglądają identycznie jak w poprzedniej sekcji.<sup>10</sup>

#### **i** ZADANIA

1. Dobierzcie się w dwuosobowe grupy.
2. Celem pierwszej osoby jest stworzenie nowego repozytorium na GitHubie, połączenie go z nowym projektem RStudio, dodanie nowego pliku `README.md` do repozytorium zawierającego tytuł wiersza "Słoń Trabalski", oraz przesłanie go na zdalne repozytorium.
3. Pierwsza osoba następnie dodaje uprawnienia do repozytorium dla drugiej osoby.
4. Druga osoba łączy się z repozytorium, dodaje do pliku `README.md` pierwszą linię wiersza (<https://wolnelektury.pl/katalog/lektura/tuwim-slom-trabalski.html>), zatwierdza zmiany oraz przesyła je na zdalne repozytorium.

## 3.6 Problemy z kontrolą wersji

W ramach jednego projektu często posiadamy wiele plików z długą historią zmian, do tego nanoszonych przez szereg różnych osób. Jest to sytuacja w której dość prosto o wystąpienie problemów czy nieoczekiwanych (przez użytkownika) zachowań systemu kontroli wersji Git.

Jednym z najczęstszych problemów jest pojawienie się poniższego komunikatu podczas próby wysyłania zmian do zdalnego repozytorium.

<sup>10</sup>Możliwe jest też dodanie systemu kontroli wersji do istniejącego projektu oraz przypisanie do niego zdalnego repozytorium, ale wymaga to większej wiedzy na temat systemu Git.

```
>>> git push
To https://github.com/YOU/REPO.git
! [rejected]          main -> main (fetch first)
error: failed to push some refs to 'https://github.com/YOU/REPO.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Oznacza on, że w repozytorium zdalnym są jakieś zmiany, których nie ma lokalnie. Prawdopodobnie wynikają one z kwestii, że inna osoba przesłała swoje zmiany do zdalnego repozytorium lub też pliki były zmienione i przesłane przez ciebie na innym komputerze. Najczęściej w takiej sytuacji wystarczy aktualizowanie zmian ze zdalnego repo (ikona strzałki w dół), a następnie ponowienie próby wysłania zmian. Czasem jednak mogły zajść zmiany w tym samym pliku edytowanym przez wiele osób. Wówczas konieczne jest ręczne poprawienie problematycznych plików, dodanie zmian i ich zatwierdzenie.

## **i** ZADANIA

1. Kontynuujcie pracę w dwuosobowych grupach.
2. Tym razem zarówno pierwsza, jak i druga osoba mają edytować plik `README.md`: pierwsza dodaje drugą linię wiersza, a druga trzecią.
3. Następnie najpierw pierwsza osoba przesła zmiany na zdalne repozytorium, a potem druga osoba próbuje to zrobić.
4. Rozwiążcie problem, który się pojawił.

*Po wykonaniu zadań będziemy mieć czas na dyskusję i pomoc.*

## **i** Komentarz

GitHub nie jest jedynym serwisem internetowym oferującym hosting repozytoriów Git. Innymi popularnymi serwisami są [GitLab](#) i [Bitbucket](#). Każdy z nich oparty jest na podobnych zasadach działania, ale różni się wieloma szczegółami. Dodatkowo możliwe jest [stworzenie własnego hostingu repozytoriów Git](#), ale wymaga to większej wiedzy technicznej.

## 4 Tworzenie pakietów R (2)

### 4.1 Publikowanie pakietów

Nowo utworzony pakiet w R można od razu umieścić na wybranym serwisie internetowym wspierającym kontrolę wersji takim jak GitHub, GitLab, czy BitBucket, gdzie nazwa repozytorium będzie identyczna jak nazwa pakietu.

Można to zrobić na kilka sposobów, na przykład:

1. Poprzez stworzenie nowego, pustego repozytorium na wybranym serwisie internetowym i sklonowanie go na swój komputer, a następnie przesłanie plików pakietu do tego repozytorium (zobacz Sekcja 3.5.1)
2. Poprzez zainicjowanie repozytorium Git używając `usethis::use_git()`, a następnie wysłanie pakietu do repozytorium GitHub poprzez `usethis::use_github()`<sup>1</sup>.

#### **i** ZADANIA

1. Opublikuj wcześniej stworzony pakiet na platformie GitHub używając jednej z powyższych metod.
2. Z poziomu RStudio stwórz nowy plik `README.md` zawierający nazwę oraz krótki opis tego pakietu.
3. Dodaj nowy plik `README.md` do repozytorium i opublikuj go na GitHubie.

### 4.2 Metadane pakietu

Sekcja 2.5 pokazała przykładowy plik `DESCRIPTION` zawierający metadane pakietu. Poniżej rozwiniemy kwestie kolejnych elementów opisu pakietu.

```
Package: mojpakiet
Title: Moje Funkcje Robiace Wszystko
Version: 0.0.1
Authors@R:
```

<sup>1</sup>Wymaga to jednak wygenerowania tokena - opis jak to zrobić można znaleźć pod adresem <https://debruine.github.io/tutorials/packages.html#github-access-token>

```
person(given = "Imie",
       family = "Nazwisko",
       role = c("cre", "aut"),
       email = "imie.nazwisko@example.com")
```

**Description:** Tworzenie, przeliczanie i wyliczanie wszystkiego.  
Czasami nawet więcej.

**License:** CC0

**Encoding:** UTF-8

**LazyData:** true

**RoxygenNote:** 7.2.3

### 4.2.1 Tytuł pakietu

Tytuł pakietu (Title:) w jednym krótkim zdaniu (sloganie) określa do czego służy ten pakiet.<sup>2</sup> Składa się on ze słów rozpoczynających się z dużej litery.

### 4.2.2 Wersja pakietu

Wersja pakietu (Version:) pozwala jego użytkownikom na zobaczenie, czy korzystają z aktualnej wersji pakietu. Zalecany sposób określania wersji pakietu jest stosowanie trzech liczb pierwsza.druga.trzecia, np. 0.9.1. Zmiana trzeciej liczby służy do pokazania, że zaszła niewielka zmiana w kodzie, zazwyczaj wiążąca się z naprawą małego błędu, np. 0.9.2. Druga liczba jest zmieniana podczas wydania nowej wersji pakietu, która zawiera większe zmiany w kodzie, jak naprawy poważnych błędów, czy dodanie nowych możliwości, np. 0.10.0. Zmiana pierwszej liczby sugeruje poważne zmiany w kodzie, które ale też sugeruje pewną stabilizację działania, np. 1.0.0.

#### Komentarz

Więcej o innych sposobach określania wersji pakietu można znaleźć pod adresami <https://semver.org/>, <https://www.x.org/releases/X11R7.7/doc/xorg-docs/Versions.html>, <https://r-pkgs.org/lifecycle.html#sec-lifecycle-version-number>.

### 4.2.3 Autorzy

Authors@R: określa kolejne osoby zaangażowane w budowę tego pakietu. W powyższym przykładzie mamy wymienioną jedną osobę "Imie" "Nazwisko", której adres mailowy to "imie.nazwisko@example.com". Dodatkowo ta osoba posiada dwie role przy tworzeniu tego pakietu "cre" oraz "aut". Pierwsza rola, "cre", informuje że ta osoba jest twórcą i

---

<sup>2</sup>Tytuły pakietów można znaleźć, np. w panelu "Packages" w RStudio.

konserwatorem tego pakietu. Ona jest odpowiedzialna za pracę pakietu. Druga rola, "aut", jest nadawana osom, które wniosły bardzo duży wkład w kod zawarty w pakiecie. Inne często używane role to "ctb" określająca osoby, które wniosły mniejszy wkład w kod (np. drobne zmiany) oraz "cph" określająca osoby czy instytucje będące posiadaczami praw autorskich (np. firma zatrudniająca autora kodu albo autor biblioteki, która została wewnętrznie użyta).<sup>3</sup> Dodanie kolejnych osób odbywa się poprzez łączenie ich funkcją `c()`.

```
Authors@R: c(
  person("Imie", "Nazwisko", role = c("cre", "aut"),
    email = "email1@example.com"),
  person("Imie2", "Nazwisko2", role = "aut", email = "email2@example.com")
)
```

#### 4.2.4 Opis pakietu

Opis pakietu (Description:) jest dłuższym (co najmniej dwuzdaniowym) opisem tego do czego służy pakiet. Powinien on też, jeżeli to możliwe, odnosić się do publikacji naukowej, która opisuje metodę używaną w pakiecie.

Przykładowy [opis pakietu](#):

```
Description: Creates superpixels based on input spatial data.
This package works on spatial data with one variable
(e.g., continuous raster), many variables (e.g., RGB rasters),
and spatial patterns (e.g., areas in categorical rasters).
It is based on the SLIC algorithm (Achanta et al. (2012)
<doi:10.1109/TPAMI.2012.120>), and readapts it to work with arbitrary
dissimilarity measures.
```

#### 4.2.5 Licencja

Licencja (License:) określa warunki korzystania z pakietu przez inne osoby. W bardzo dużym skrócie licencje oprogramowania można podzielić na licencje otwarte (*open-source*) oraz zamknięte (*proprietary*). Najpopularniejsze licencje otwarte używane w pakietach R to licencja CC0, MIT oraz GPL. Pierwsza z nich, CC0 oznacza przekazanie [zawartości pakietu do domeny publicznej](#) i najczęściej stosowana jest do pakietów zawierających tylko zbiory danych. Licencja MIT daje nieograniczone prawo do używania, modyfikowania i rozpowszechniania kodu, pod warunkiem zachowania informacji o autorze. Dodanie licencji MIT do pakietu R można wykonać podając swoje imię i nazwisko w funkcji `usethis::use_mit_license("Imie Nazwisko")`. W ten sposób informacja o tej licencji zostanie dodana do pliku DESCRIPTION

---

<sup>3</sup>Pełną listę dostępnych ról można znaleźć pod adresem <http://www.loc.gov/marc/relators/relaterm.html>.

(License: MIT + file LICENSE) oraz zostaną utworzone specjalne pliki z treścią licencji. Trzecia z licencji otwartych, GPL (ang. *GNU General Public License*) pozwala użytkownikom na uruchamianie, dostosowywanie, rozpowszechnianie i udoskonalanie kodu. Ważną cechą tej licencji jest wymaganie, że wszelkie prace oparte o kod w licencji GPL również muszą mieć licencję GPL. Oprogramowanie zamknięte może również przyjmować wiele form (np. freeware czy też oprogramowanie komercyjne). Określenie pakietu jako oprogramowania zamkniętego odbywa się poprzez dodanie informacji, że licencja znajduje się w pliku LICENSE (License: file LICENSE), a następnie stworzenie pliku tekstowego o tej nazwie zawierającego odpowiednią modyfikację poniższego tekstu:

```
Proprietary
```

```
Do not distribute outside of NAZWA MOJEJ FIRMY.
```

#### Komentarz

Więcej o licencjach można przeczytać na stronach <http://choosealicense.com/licenses/> oraz <https://tldrlegal.com/>.

### 4.2.6 Inne pola

W pliku DESCRIPTION można również określić inne pola, np. LazyData czy SystemRequirements. Możliwe jest nawet dodawanie własnych pól. Więcej na ten temat można przeczytać pod adresem <https://r-pkgs.org/description.html#other-fields>.

#### ZADANIA

1. Uzupełnij opis swojego pakietu poprzez dodanie informacji o tytule, wersji, autorach, opisie i licencji w pliku DESCRIPTION.
2. Prześlij zmiany do repozytorium na GitHubie.

## 4.3 Dokumentacja pakietu

Po wykonaniu poprzednich kroków posiadamy działający pakiet, którego funkcje posiadają odpowiednią dokumentację. Teraz konieczne jest stworzenie dokumentacji pakietu - ma ona na celu poinformować potencjalnych użytkowników do czego pakiet służy, jak go zainstalować, czy też pokazać przykłady jego użycia. Pakiety mogą być dokumentowane używając kilku różnych rodzajów plików, np. za pomocą pliku README.Rmd, tzw. winiety (ang. *vignette*), czy pliku NEWS.md. Każdy z nich ma swój cel.

### 4.3.1 README

Plik `README.Rmd` można stworzyć za pomocą funkcji `usethis::use_readme_rmd()`. W efekcie będzie się on znajdował w głównym folderze pakietu. Ten plik powinien zawierać:<sup>4</sup>

1. Nazwę pakietu
2. Opis do czego pakiet służy
3. Instrukcje jak go zainstalować
4. Prosty przykład użycia
5. Odnośniki do podobnych prac, programów, czy artykułów naukowych

Warto tutaj zrozumieć różnicę między plikami `README.md` i `README.Rmd`. `README.md` jest plikiem w formacie Markdown, który jest automatycznie wyświetlany na stronie repozytorium na GitHubie. `README.Rmd` jest plikiem w formacie RMarkdown, który może być przetworzony do formatu `.md`. Użycie RMarkdown pozwala na dodanie kodu R, który zostanie automatycznie przetworzony do wyników jego działania – w efekcie konwersji pliku `.Rmd` otrzymamy plik `.md`. Typowe użycie pliku `README.Rmd` polega na dodaniu do niego zmian, a następnie przetworzeniu go do formatu `.md` (ikona knit) i przesłaniu na GitHuba.

#### Komentarz

Markdown to prosty język znaczników służący do formatowania tekstu. Jego idea polega na tym, że w dokumencie tekstowym używamy specjalnych znaków, które po przetworzeniu dokumentu wyświetlają się w odpowiedni sposób. Przykładowo jedna gwiazdka przed tekstem i jedna po tekście oznacza pochylony tekst (*\*pochylony tekst\**), a dwie gwiazdki przed i po oznaczają pogrubiony tekst (**\*\*pogrubiony tekst\*\***). Innym przykładem są nagłówki określone poprzez jeden lub więcej symboli kratki.

```
# Nagłówek
```

```
## Nagłówek drugiego poziomu (mniejsza czcionka)
```

Zestawienie pokazujące podstawy składni RMarkdown jest wbudowane w RStudio i można je wyświetlić za pomocą `Help -> Markdown Quick Reference` lub pod adresem [https://rmarkdown.rstudio.com/authoring\\_basics.html](https://rmarkdown.rstudio.com/authoring_basics.html).

### 4.3.2 Winiety

Winiety mają na celu pokazanie bardziej złożonego przykładu użycia pakietu. Nową winietę można stworzyć za pomocą funkcji `usethis::use_vignette("nazwa-winiety")`. W tym momencie zostanie stworzony nowy plik `nazwa-winiety.Rmd` w folderze `vignettes`. Teraz

<sup>4</sup>Dodatkowe elementy to oznaki (ang. *badges*) pokazujące, np. status pakietu, liczbę jego pobrań i wiele innych.



możliwe jest jego edytowanie i dodawanie nowej treści. Pakiety mogą posiadać wiele różnych winiet, zawierających coraz bardziej zaawansowane przykłady lub też opis różnych grup funkcji z pakietu.

#### Komentarz

Pliki RMarkdown mogą być przetworzone (ang. *render*) do wielu różnych formatów plików, między innymi html, pdf, czy word w zależności od określonych opcji w nagłówku pliku. To przetworzenie może odbyć się używając ikony “Knit” w RStudio lub funkcji `rmarkdown::render()`.

### 4.3.3 NEWS

Elementem dokumentowania pakietu jest również informowanie o tym jakie nowe zmiany zaszły wraz z kolejnymi wersjami pakietu. W pakietach R może mieć to miejsce używając pliku `NEWS.md` tworzonego poprzez `usethis::use_news_md()`. Taki plik może zawierać informacje o nowych funkcjach, zmianach istniejących funkcji, naprawionych błędach, itd. Przykład szablonu pliku `NEWS.md` można znaleźć pod adresem [https://ropensci.github.io/dev\\_guide/newstemplate.html](https://ropensci.github.io/dev_guide/newstemplate.html).

#### ZADANIA

1. Stwórz plik `README.Rmd` zawierający opis swojego pakietu, instrukcje instalacji i przykładowe jego użycie.
2. Przetwórz ten plik do formatu `.md` i prześlij go do repozytorium na GitHubie. Obejrzyj go na stronie internetowej repozytorium.
3. Stwórz prostą winietę zawierającą jeden przykład użycia swojego pakietu.
4. Dodaj pierwsze informacje o zmianach w pakiecie do pliku `NEWS.md`.
5. Wyślij nowe pliki do repozytorium na GitHubie.

## 5 Kontrola wersji (2)

### 5.1 Git

Do tej pory używaliśmy systemu kontroli wersji Git z poziomu RStudio. Każda z ikon na pasku narzędziowym RStudio odpowiada jednej z komend systemu Git. W rzeczywistości, system Git jest niezależny od RStudio i może być używany z poziomu konsoli systemowej. Git składa się z kilkudziesięciu komend, których działanie jest dalej uzależnione od podanych argumentów. Tutaj przedstawiony zostanie tylko podzbiór najczęściej używanych. Pełniejszy opis komend systemu Git można znaleźć pod adresem <https://education.github.com/git-cheat-sheet-education.pdf> lub <http://rogerdudler.github.io/git-guide/index.pl.html>.

### 5.2 Repozytorium

Podstawowym elementem systemu Git jest repozytorium (ang. *repository*, często określane skrótowo jako *repo*). Jest to folder, który przechowuje wszystkie pliki i foldery w ramach jednego projektu.<sup>1</sup> Dodatkowo wewnątrz repozytorium znajduje się ukryty folder `.git`, który zawiera informacje o historii i zmianach każdego z naszych plików. Repozytorium może znajdować się na dysku naszego komputera (wtedy jest nazywane repozytorium lokalnym) lub też na serwerze w internecie (określane jako repozytorium zdalne (ang. *remote*)).

```
# określenie obecnego katalogu jako repozytorium Git
git init
```

### 5.3 Dodawanie zmian

W nowo utworzonym repozytorium możemy tworzyć nowe pliki oraz edytować już istniejące. Po pewnym czasie możemy stwierdzić, że dodaliśmy nową funkcjonalność do funkcji lub naprawiliśmy błąd w kodzie. Wtedy należy, po zapisaniu również pliku na dysku, dodać te zmiany do systemu Git. Po dodaniu zmian są one przechowywane w miejscu określanym jako *Index*. Działa ono jak poczekalnia - w tym momencie zmiany jeszcze nie są potwierdzone, ale możemy sprawdzić co zmieniło się od ostatniego zatwierdzenia zmian.

---

<sup>1</sup>W kontekście R, warto o tym myśleć jako o projekcie RStudio.

```
# dodanie pojedynczego pliku
git add sciezka_do_pliku
# dodanie wszystkich plików
git add --all
```

## 5.4 Sprawdzanie zmian

Zanim zatwierdzimy zmiany można je sprawdzić. W ten sposób dla każdej linii tekstu (kodu) otrzymuje się informacje co zostało dodane lub usunięte.

```
# sprawdzenie dodanych zmian
git diff
```

## 5.5 Zatwierdzanie zmian

Zatwierdzanie zmian (ang. **commit**) powoduje ich zapisanie na stałe w systemie Git. Wymaga to dodania wiadomości, która opisuje wprowadzone zmiany.

```
# zatwierdzenie dodanych zmian
git commit -m "opis wprowadzonych zmian"
```

## 5.6 Rozgałęzienia

Częstą sytuacją jest posiadanie stabilnego, działającego kodu, ale co do którego mamy pomysły jak go ulepszyć, np. zwiększyć jego wydajność. Wtedy edycja poprawnego kodu może nie przynieść najlepszych wyników - co jeżeli nasz pomysł się jednak nie sprawdzi? Lepszą możliwością jest użycie rozgałęzień (ang. *branches*) w systemie Git. Domyślnie nowe repozytorium posiada już jedną gałąź nazwaną main.

```
# wypisanie wszystkich rozgałęzień
git branch
```

Kolejnym krokiem jest utworzenie nowego rozgałęzienia. W efekcie tego działania nowa gałąź staje się odniesieniem do istniejącego stanu obecnej gałęzi.

```
# utworzenie nowego rozgałęzienia
git branch nazwa_nowej_galezi
```

Co ważne utworzenie nowego rozgałęzienia nie powoduje przejście do niego - należy to samodzielnie wykonać.

```
# przejście do innego rozgałęzienia
git checkout nazwa_nowej_galezi
```

W tym momencie możliwe jest testowanie różnych możliwości ulepszenia istniejącego kodu bez obawy, że wpłynie to na jego działającą wersję. Po stwierdzeniu, że nasze zmiany są odpowiednie należy je dodać i zatwierdzić. Teraz można powrócić do głównej gałęzi (main) i dołączyć zmiany stworzone w innej gałęzi.

```
# powrót do głównej gałęzi
git checkout main
# połączenie wybranego rozgałęzienia z obecnym
git merge nazwa_nowej_galezi
```

#### Komentarz

Tworzenie nowych gałęzi i przechodzenie między nimi jest też możliwe w RStudio. W tym celu należy wybrać ikonę z napisem “New branch” albo nazwę obecnej gałęzi.

## 5.7 Repozytorium zdalne

System Git ma wiele zalet w przypadku samodzielnej pracy na własnym komputerze, zyski z jego używania są jednak znacznie większe, gdy nasze repozytoria mają też zdalne odpowiedniki.

Łączenie się ze zdalnymi repozytoriami może nastąpić na dwa sposoby. W pierwszym z nich repozytorium zdane już istnieje, a my chcemy się do niego podłączyć i je pobrać.

```
# pobranie kopii istniejącego zdalnego repo
git clone sciezka_do_zdalnego_repo
```

Drugim sposobem jest posiadanie istniejącego, lokalnego repozytorium, a następnie dodanie do niego adresu zdalnego repozytorium.

```
# dodanie ścieżki do zdalnego repo
git remote add origin sciezka_do_zdalnego_repo
```

## 5.8 Wysyłanie zmian

Obecne dodane i zatwierdzone zmiany znajdują się jedynie w repozytorium lokalnym. Konieczne jest ich wysłanie do zdalnego repozytorium.

```
# wysyłanie zmian do zdalnego repo  
git push
```

## 5.9 Aktualizowanie zmian

Zdalne repozytoria mogą pozwalać na nadawanie różnych uprawnień użytkownikom. Możliwe jest określenie, że inne osoby mogą nanosić zmiany w zdalnych repozytoriach. Dodatkowo, jedna osoba może zmieniać zdalne repozytoria używając różnych komputerów. Konieczne jest więc aktualizowanie zmian, które zaszyły w zdalnym repozytorium na lokalnym komputerze.

```
# aktualizowanie zmian ze zdalnego repo  
git pull
```

### ZADANIA

1. Dobierzcie się w dwuosobowe grupy: każda z osób będzie pracowała na repozytorium pakietu drugiej osoby.
2. Używając linii komend, sklonujcie repozytorium pakietu drugiej osoby.
3. Stwórzcie nową gałąź w repozytorium i przejdźcie do niej.
4. Dodajcie do repozytorium plik z kodem R, który będzie zawierał funkcję, która zwraca sumę dwóch liczb.
5. Sprawdźcie czy pakiet spełnia wszystkie wymagania i działa poprawnie.
6. Prześlijcie nowe zmiany do zdalnego repozytorium używając linii komend. Co ważne – nie łączcie ich z główną gałęzią! (W razie problemów upewnijcie się, że druga osoba ma uprawnienia do wprowadzania zmian w repozytorium.)
7. Obejrzyjcie zmiany w repozytorium zdalnym.

## 6 Tworzenie pakietów R (3)

### 6.1 Dane w pakietach R

Pakiety R mogą również zawierać dane. Zazwyczaj takie dane służą do trzech celów: (1) do przechowywania danych, które są używane wewnątrz pakietu, (2) do użycia w przykładach i innych formach dokumentacji, oraz (3) do testowania funkcji.

Istnieje kilka mechanizmów do przechowywania danych w pakietach R:

1. Dane w postaci binarnej w folderze `data/`
2. Dane w postaci oryginalnej w podfolderze folderu `inst/`
3. Dane w postaci binarnej w pliku `R/sysdata.rda`
4. Dane stworzone wewnątrz kodu R, np. kodu przykładu czy kodu testu

Każdy z nich ma swoje wady i zalety, szczególnie widoczne w przypadku wykorzystywania danych przestrzennych.

#### Komentarz

CRAN z reguły nie pozwala na umieszczanie w pakietach danych o rozmiarze większym niż ok. 5 MB. W związku z tym, jeżeli dane są większe niż 5 MB, to należy je umieścić w innym miejscu, np. na GitHubie, a wewnątrz pakietu umieścić tylko kod do ich pobrania.

#### 6.1.1 Dane w postaci binarnej w folderze `data/`

Niemal każdy obiekt R można zapisać wewnątrz pakietu jako plik binarny (`.rda`) w folderze `data/`. Najprościej dodać taki obiekt używając funkcji `usethis::use_data()`:

```
moj_df = data.frame(x = 1:10, y = 11:20)
usethis::use_data(moj_df, overwrite = TRUE)
```

Powyższy kod stworzy plik `data/moj_df.rda` zawierający obiekt `moj_df`. Teraz ten obiekt jest dostępny wewnątrz pakietu i można go użyć wewnątrz funkcji, przykładów, czy testów jednostkowych.

**i**

Jeden z możliwych elementów opisu pakietu w pliku DESCRIPTION to LazyData. Przyjmuje on wartość `true` lub `false` i określa czy dane w folderze `data/` są ładowane do pamięci podczas ładowania pakietu. W przypadku `LazyData: true` dane są ładowane do środowiska pracy podczas ładowania pakietu i są od razu dostępne. W przypadku `LazyData: false` dane są ładowane do środowiska pracy dopiero w momencie ich wywołania, np. poprzez funkcję `data()`.

Warto kod R użyty do stworzenia takich danych umieścić w folderze `data-raw/`. A sam folder `data-raw/` należy dodać do pliku `.Rbuildignore`.<sup>1</sup> Dzięki temu możliwe będzie odtworzenie budowy takich danych w przyszłości, jeżeli zajdzie taka konieczność.

Dane załączone w ten sposób muszą być udokumentowane. Taka dokumentacja to plik o rozszerzeniu .R w folderze R/ zawierający funkcję roxygen2 z odpowiednimi tagami. Przykładowo, dla obiektu `moj_df` można stworzyć plik `R/moj_df.R` zawierający:

```
#' Example data  
#'  
#' Example data created manually for the purpose of this package.  
#'  
#' @format ## `moj_df`  
#' A data frame with 10 observations on 2 variables.  
#' \describe{  
#'   \item{x}{numeric vector}  
#'   \item{y}{numeric vector}  
#' }  
#' @source Manually created.
```

"moj\_df"

### 6.1.2 Dane w postaci oryginalnej w podfolderze folderu inst/

Wewnątrz podfolderu folderu `inst/` (np. `inst/data/`) można umieścić dowolne pliki, które będą dostępne po zainstalowaniu pakietu. Dotyczy to, np., plików `.csv` ale także różnorodnych formatów GIS, `.gpkg` czy `.tif`.

Po instalacji pakietu, folder `inst/` nie jest dostępny, a jego zawartość znajduje się bezpośrednio w folderze instalacyjnym pakietu. Dostępne foldery można sprawdzić używając funkcji `system.file()` oraz `dir()`.

```
system.file(package = "spData") |> dir()
```

<sup>1</sup>Można też użyć funkcji `usethis::use_data_raw()`.

```
[1] "data"          "DESCRIPTION" "help"          "html"          "INDEX"
[6] "Meta"          "misc"         "NAMESPACE"     "R"             "raster"
[11] "README"        "shapes"        "weights"
```

W tym przypadku mamy kilka folderów z danymi: data/, misc/, raster/, shapes/, oraz weights/. Sprawdzenie zawartości danego folderu jest możliwe poprzez podanie jego nazwy jako argumentu funkcji `system.file()`.

```
system.file("raster", package = "spData") |> dir()
```

```
[1] "elev.tif"          "grain.tif"          "grain.tif.aux.xml"
```

Następnie można użyć funkcji `system.file()` do określenia ścieżki do wybranego pliku na danym komputerze.

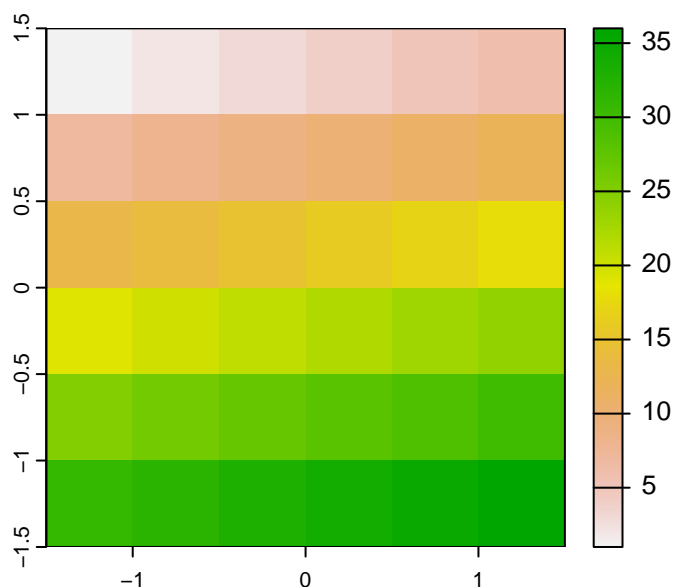
```
system.file("raster/elev.tif", package = "spData")
```

```
[1] "/home/jn/R/x86_64-redhat-linux-gnu-library/4.3/spData/raster/elev.tif"
```

Znajomość tej ścieżki pozwala na wczytanie pliku do R.

```
library(terra)
elev = rast(system.file("raster/elev.tif", package = "spData"))
plot(elev)
```





#### **i** Komentarz

Przechowywanie danych w postaci binarnej jest szczególnie dobrym rozwiązaniem w przypadku, gdy przechowywane obiekty mają jedną z klas wbudowanych w język R (np. ramka danych czy macierz).

Podejście oparte na przechowywaniu danych w postaci oryginalnej jest szczególnie dobre w kontekście danych przestrzennych:

1. Interpretacja elementów obiektów przestrzennych (np. układu współrzędnych przez bibliotekę PROJ) zmienia się w czasie. W związku z tym, dane przestrzenne zapisane w postaci binarnej mogą być niepoprawne w przyszłości.
2. Główna klasa obiektów w pakiecie **terra** nie przechowuje wartości, ale tylko wskaźnik (ang. *pointer*) C++ do pliku na dysku. W związku z tym, obiekty **terra** zapisane w postaci binarnej mogą nie być poprawnie załadowane do pamięci podczas ładowania pakietu.
3. Dane są dostępne w oryginalnym formacie, co umożliwia ich użycie poza R.

### 6.1.3 Dane w postaci binarnej w pliku R/sysdata.rda

Wewnątrz pakietu można umieścić dane w postaci binarnej w pliku R/sysdata.rda. Takie dane są dostępne tylko wewnątrz pakietu i nie są ładowane do środowiska pracy. Nie wymagają też dodatkowej dokumentacji.

Do stworzenia takiego pliku użyć funkcji `usethis::use_data()` z argumentem `internal = TRUE`. Należy tylko pamiętać, że plik `R/sysdata.rda` może zawierać więcej niż jeden obiekt i wówczas te obiekty należy zapisać w poniższy sposób.

```
moj_df = data.frame(x = 1:10, y = 11:20)
moj_df2 = data.frame(x = 1:10, y = 11:20)
usethis::use_data(moj_df, moj_df2, internal = TRUE)
```

#### 6.1.4 Dane stworzone wewnątrz kodu R

Małe dane można stworzyć bezpośrednio wewnątrz kodu R. Może to mieć miejsce przed przykładem użycia danej funkcji czy też przed testem jednostkowym.

##### ZADANIA

1. Stwórz nowy obiekt zawierający dane przykładowe, które będzie można użyć w przykładach twojego pakietu.
2. Zapisz kod tworzący ten obiekt do folderu `data-raw/`.
3. Dodaj ten obiekt do folderu `data/` używając funkcji `usethis::use_data()`.
4. Dodaj dokumentację dla tego obiektu do folderu `R/`.
5. Zapisz ten obiekt do zewnętrznego formatu (np. `.csv`) i umieść go wewnątrz podfolderu folderu `inst/`.
6. Użyj zarówno wewnętrznych jak i zewnętrznych danych wewnątrz przykładu jakieś funkcji z twojego pakietu.
7. Uruchom `devtools::document()` a następnie `devtools::check()` i sprawdź czy wszystko działa poprawnie.

## 6.2 Testy jednostkowe

Testy jednostkowe (ang. *unit tests*) to sposób sprawdzania czy stworzona przez nas funkcja działa w sposób jaki oczekujemy. Tworzenie takich testów wymusza także myślenie na temat odpowiedniego działania funkcji i jej API. Testy jednostkowe są najczęściej stosowane w przypadku budowania pakietów, gdzie możliwe jest automatyczne sprawdzenie wielu testów na raz. Przykładowo, napisaliśmy nową funkcję, która wykonuje złożone operacje i, po wielu sprawdzeniach, wiemy, że daje poprawne wyniki. Po kilku miesiącach wpadliśmy na pomysł jak zwiększyć wydajność naszej funkcji. W tym momencie wystarczy już tylko stworzyć nową implementację i użyć wcześniej zbudowanych testów. Dadzą one informację, czy efekt działania jest taki jaki oczekujemy, a w przeciwnym razie wskażą gdzie pojawił się błąd. Istnieje też dodatkowa reguła - jeżeli znajdziesz błąd w kodzie od razu napisz test jednostkowy.

Zobaczmy jak działają testy jednostkowe na przykładzie funkcji `powierzchnia()`, która oblicza powierzchnię prostokąta na podstawie macierzy zawierającej współrzędne jego wierzchołków.

```
powierzchnia = function(x){  
  a = x[1, 2] - x[1, 1]  
  b = x[2, 2] - x[2, 1]  
  a * b  
}
```

Jednym z możliwych narzędzi do testów jednostkowych w R jest pakiet **testthat** (Wickham 2020).

```
library(testthat)
```

Zawiera on szereg funkcji sprawdzających czy działanie naszych funkcji jest zgodne z oczekiwaniem. Funkcje w tym pakiecie rozpoczynają się od prefiksu `expect_` (oczekuj).

W przypadku funkcji `powierzchnia()` oczekujemy, że wynik będzie zawierał tylko jeden element. Możemy to sprawdzić za pomocą funkcji `expect_length()`.

```
nowy_p = matrix(c(0, 0, 6, 5), ncol = 2)  
expect_length(powierzchnia(nowy_p), 1)
```

Jeżeli wynik ma długość jeden to wówczas nic się nie stanie. W przeciwnym razie pojawi się komunikat błędu.

Wiemy, że `powierzchnia` naszego przykładowego obiektu `nowy_p` to 30. Do sprawdzenia, czy nasza funkcja daje na tym obiekcie dokładnie taki wynik możemy użyć `expect_equal()`.

```
expect_equal(powierzchnia(nowy_p), 30)
```

W momencie, gdy wynik jest zgodny to nie nastąpi żadna reakcja, a w przeciwnym razie wystąpi błąd. W pakiecie **testthat** istnieją inne funkcje podobne do `expect_equal()`. Przykładowo, funkcja `expect_identical()` sprawdza nie tylko podobieństwo wartości, ale też to czy klasa wyników jest taka sama.

Aby sprawdzić czy nasza funkcja na pewno zwróci błąd w przypadku podania niepoprawnych danych wejściowych możemy użyć funkcji `expect_error()`. Jej działanie jest przedstawione poniżej.

```
expect_error(nowy_prostokat(3, 5, 2, "a"))  
expect_error(nowy_prostokat(1, 2, 3, 6))
```

W przypadku, gdy wywołanie funkcji zwróci błąd, `expect_error()` nic nie zwróci. Natomiast, jeżeli wywołania funkcji nie zwróci błędu, `expect_error()` zatrzyma swoje działanie i zwróci komunikat. Odpowiednikami `expect_error()` dla ostrzeżeń jest `expect_warning()`, a dla wiadomości `expect_message()`.

Pozostałe funkcje z tego pakietu są wymienione i opisane na stronie <https://testthat.r-lib.org/reference/index.html>.

### **i** ZADANIA

1. Stwórz nowy plik `temp.R`.
2. Wewnątrz tego pliku dodaj kilka testów sprawdzających funkcję ze swojego pakietu.
3. Upewnij się, że wszystkie testy działają poprawnie (nie zwracają błędów).

## 6.3 Wbudowane testy

Testy jednostkowe można również wbudować wewnątrz pakietu - w efekcie, gdy naniesiemy w nim jakieś zmiany możemy sprawdzić czy otrzymujemy takie same wyniki.

Pierwszym krokiem do używania wbudowanych testów jest ustawienie odpowiedniej infrastruktury używając funkcji `use_testthat()`. Powoduje ona dodanie pakietu **testthat** do wpisu `Suggests`:, stworzenie folderów `tests/` i `tests/testthat/` oraz pliku `tests/testthat.R`.

```
use_testthat()
#> [?] Adding 'testthat' to Suggests field in DESCRIPTION
#> [?] Creating 'tests/testthat/'
#> [?] Writing 'tests/testthat.R'
```

Teraz możliwe jest napisanie testów jednostkowych. Zazwyczaj polega to na stworzeniu oddzielnego pliku dla każdej funkcji z naszego pakietu. Przykładowo, nasz pakiet zawiera funkcję `powierzchnia()`, dlatego też do jego testowania możemy stworzyć nowy plik `tests/testthat/test-powierzchnia.R`. Wewnątrz tego pliku należy sprawdzać kolejne aspekty działania kodu używając funkcji `test_that()`, gdzie należy podać (1) opis tego co jest sprawdzane i (2) testy wewnątrz nawiasów klamrowych. Przykładowy plik `tests/testthat/test-powierzchnia.R` może wyglądać w ten sposób:

```
nowy_p = matrix(c(0, 0, 6, 5), ncol = 2)

test_that("struktura wyniku jest poprawna", {
  expect_length(powierzchnia(nowy_p), 1)
```

```

}))

test_that("wartosc wyniku jest poprawna", {
  expect_equal(powierzchnia(nowy_p), 30)
})

test_that("wystepuja odpowiednie bledy", {
  expect_error(nowy_prostokat(3, 5, 2, "a"))
})

```

Po napisaniu testów można sprawdzić czy wszystkie z nich dają odpowiedni wynik używając `devtools::test()`<sup>2</sup>. W efekcie wyświetlone zostaną wszystkie testy i zostanie wskazane, które z nich się nie powiodły i należy je poprawić.

## 6.4 Pokrycie kodu testami jednostkowymi

W przypadku testów jednostkowych możliwe jest sprawdzenie pokrycia kodu testami. Pokrycie kodu to procent kodu, który jest wykonywany podczas uruchamiania testów. W przypadku, gdy pokrycie kodu jest niskie, oznacza to, że testy nie sprawdzają wszystkich możliwych przypadków działania funkcji. Inaczej mówiąc, może to oznaczać, że pewne części kodu nie są testowane i mogą zawierać błędy lub działać niepoprawnie.

Pokrycie kodu testami można sprawdzić lokalnie używając funkcji `devtools::test_coverage()` lub też można użyć systemu CI/CD (o tym później), żeby sprawdzić pokrycie testami na zewnętrznym serwerze.

### **i** ZADANIA

1. Przenieś testy z pliku `temp.R` do odpowiedniego pliku wewnątrz folderu `tests/testthat/`.
2. Usuń plik `temp.R`.
3. Uruchom testy używając `devtools::test()`. Czy wszystkie z nich działają poprawnie?
4. Spróbuj użyć danych wbudowanych w pakiet do testów.
5. Uruchom `devtools::test_coverage()` i sprawdź pokrycie kodu testami.

<sup>2</sup>Testy są też automatycznie uruchamiane podczas sprawdzania pakietu

## 6.5 Strona internetowa pakietu

Dodatkowo w R istnieje możliwość prostego stworzenia stron internetowych dla wybranego pakietu używając pakietu **pkgdown** (Wickham i Hesselberth 2020). Przykład takiej strony można zobaczyć pod adresem <https://pkgdown.r-lib.org/index.html>.

Stworzenie strony pakietu wymaga jedynie wywołania funkcji `usethis::use_pkgdown()` wewnątrz pakietu R. W efekcie działania tej funkcji zostanie utworzony folder `docs/` zawierający stronę internetową reprezentującą pakiet i jego dokumentację. W przypadku, gdy pakiet znajduje się na GitHubie możliwe jest wyświetlenie tej strony pod adresem `https://<nazwauzytkownika>.github.io/<nazwapakietu>/`. Aby ta strona była dostępna w internecie należy na platformie GitHub wejść w zakładkę Settings, następnie znaleźć część określoną jako Pages, i określić Source jako “main branch /docs folder”.

Alternatywą dla powyższej funkcji jest `usethis::use_pkgdown_github_pages()`. Wykonuje ona więcej kroków automatycznie: tworzy nową gałąź Git oraz dodaje nowy plik konfiguracji GitHub Action (więcej o tym w kolejnej sekcji), który będzie starał się automatycznie budować stronę internetową pakietu w chmurze po każdej zmianie w kodzie.

## 6.6 CI/CD

Ciągła integracja i wdrażanie (ang. *continuous integration and deployment*, CI/CD) to podejście do tworzenia oprogramowania, które polega na łączeniu zmian w kodzie, a następnie automatycznym testowaniu tych zmian. Jest to szerokie podejście, które może być realizowane na wiele sposobów.

Tutaj skupimy się na paru przykładach CI/CD powiązanych z pracą nad pakietami R. Ten temat jest również mocno powiązany z kontrolą wersji oraz platformami do hostowania kodu, takimi jak GitHub, GitLab, czy Bitbucket.

Platforma GitHub oferuje CI/CD poprzez system o nazwie [GitHub Actions](#). Przykłady działania GitHub Actions można znaleźć w zakładce Actions na stronie każdego repozytorium. W dużym skrócie, GitHub Actions pozwala na uruchamianie kodu w chmurze po przesłaniu zmiany w kodzie na platformę GitHub.<sup>3</sup> Najprościej można wyobrazić sobie ten proces jako przesłanie kodu na chmurę należącą do GitHuba, gdzie zostaje on uruchomiony i przetestowany.<sup>4</sup>

W przypadku pakietów R, GitHub Actions może być używany do automatycznego budowania i sprawdzania pakietu (na różnych systemach operacyjnych), wykonywania testów jednostkowych, sprawdzania pokrycia kodu testami, oraz do automatycznego budowania strony internetowej pakietu. Samo używanie GitHub Actions jest możliwe poprzez dodanie pliku konfiguracyjnego do repozytorium. Taki plik (lub pliki) musi znajdować się w folderze `.github/workflows/` i mieć rozszerzenie `.yaml`.

<sup>3</sup>Możliwe jest też uruchamianie kodu co zadany czas.

<sup>4</sup>Należy pamiętać, że GitHub Actions nie jest w pełni darmowy.

Przykładowy plik konfiguracyjny dla GitHub Actions może wyglądać w ten sposób:

```
on:
  push:
    branches: [main]

name: R-CMD-check

jobs:
  R-CMD-check:
    runs-on: ubuntu-latest

    name: ubuntu-latest release

    env:
      GITHUB_PAT: ${ secrets.GITHUB_TOKEN }
      R_KEEP_PKG_SOURCE: yes

    steps:
      - uses: actions/checkout@v3

      - uses: r-lib/actions/setup-pandoc@v2

      - uses: r-lib/actions/setup-r@v2
        with:
          r-version: release
          http-user-agent: ubuntu-latest
          use-public-rspm: true

      - uses: r-lib/actions/setup-r-dependencies@v2
        with:
          extra-packages: any::rcmdcheck
          needs: check

      - uses: r-lib/actions/check-r-package@v2
        with:
          upload-snapshots: true
```

Dodania pliku konfiguracyjnego można wykonać ręcznie lub przy pomocy funkcji `usethis::use_github_action()`, która pozwala na wybranie jednego z kilku szablonów. Warto zacząć od wyboru jednego z szablonów i dostosowania go do własnych potrzeb.

Powyższy plik konfiguracyjny ma na celu sprawdzenie pakietu na systemie Ubuntu. Możliwe jest także użycie systemu GitHub Actions do, np. określenia pokrycia kodu

testami czy też automatycznego budowania strony internetowej pakietu. Przykłady różnych plików konfiguracyjnych związanych z pakietami R można znaleźć w repozytorium <https://github.com/r-lib/actions/tree/v2/examples>. W kontekście tworzenia pakietów R związanych z danymi przestrzennymi, warto sprawdzić pliki konfiguracyjne GitHub Actions już istniejących pakietów R opartych o biblioteki GDAL czy PROJ.

## **i** ZADANIA

1. Dodaj do swojego pakietu plik konfiguracyjny dla GitHub Actions, który będzie sprawdzał pakiet na systemie Ubuntu.
2. Prześlij zmiany na GitHuba i sprawdź czy GitHub Actions działa poprawnie.
3. Dodaj do swojego pakietu plik konfiguracyjny dla GitHub Actions, który będzie sprawdzał pokrycie kodu testami. Możesz to zrobić albo rozszerzając plik konfiguracyjny z poprzedniego zadania, albo tworząc nowy plik konfiguracyjny.
4. Prześlij zmiany na GitHuba i sprawdź czy GitHub Actions działa poprawnie.
5. Dodaj do swojego pakietu plik konfiguracyjny dla GitHub Actions, który będzie budował stronę internetową pakietu. Możesz to zrobić albo rozszerzając plik konfiguracyjny z poprzedniego zadania, albo tworząc nowy plik konfiguracyjny.
6. Prześlij zmiany na GitHuba i sprawdź czy GitHub Actions działa poprawnie oraz czy strona internetowa pakietu istnieje.
7. W przypadku, gdy wszystko działa poprawnie, dodaj dwie nowe linie do pliku DESCRIPTION zawierające linki do strony internetowej pakietu oraz do repozytorium pakietu:

```
URL: https://ja.github.io/mojpakiet/
BugReports: https://github.com/ja/mojpakiet/issues
```

## 6.7 CRAN

CRAN (*Comprehensive R Archive Network*) to repozytorium zawierające pakiety R. Wszystkie pakiety znajdujące się w tym repozytorium są sprawdzane pod kątem zgodności z wymaganiami na kilku systemach operacyjnych. Wymagania można znaleźć pod adresami <https://cran.r-project.org/web/packages/policies.html> oraz [https://cran.r-project.org/web/packages/submission\\_checklist.html](https://cran.r-project.org/web/packages/submission_checklist.html). Co ważne, wymagania te zmieniają się w czasie.<sup>5</sup>

<sup>5</sup>Niektóre zasady działania CRAN są krytykowane, co można przeczytać pod adresem [https://github.com/cranchange/cran\\_change.org](https://github.com/cranchange/cran_change.org).



#### Komentarz

Tworzenie nowych narzędzi daje często satysfakcję. Należy jednak pamiętać, że narzędzia komputerowe są zależne od innych narzędzi, które mogą się zmieniać w czasie. W związku z tym, narzędzia komputerowe wymagają stałej uwagi oraz konserwacji (ang. *maintenance*).

Podejście stosowane przez CRAN kładzie odpowiedzialność za poprawność działania pakietu na jego autorach. W zamian, potencjalni użytkownicy mogą mieć pewność, że pakiet jest zgodny z wymaganiami CRAN, działa poprawnie i jest możliwy do zainstalowania na różnych systemach operacyjnych. Dodatkowo, CRAN tworzy wersje binarne pakietów dla systemów Windows i Mac OS, co znacznie ułatwia i przyspiesza ich instalację.

#### Komentarz

Wcześniej przedstawiona funkcja `devtools::check()` sprawdza pakiet pod kątem różnych testów, ale nie sprawdza czy jest on w pełni zgodny z wymaganiami CRAN. Wynika to z kilku kwestii. Trzy najważniejsze to: 1. Niektóre z wymagań są sprawdzane manualnie przez CRAN team. 2. Niektóre z wymagań nie są w pełni znane. 3. CRAN sprawdza pakiety na różnych systemach operacyjnych, a `devtools::check()` sprawdza pakiet tylko na systemie, na którym jest uruchomiony.

W momencie gdy uważam, że pakiet jest w wersji stabilnej i gotowy do wysłania na CRAN, wykonuję następujące kroki:

```
# 1 dodaję dokumentację
devtools::document()
# 2 testuję pakiet
devtools::check()
# 3 sprawdzam czy pakiet jest zgodny z wymaganiami CRAN
rcmdcheck::rcmdcheck(args = c("--no-manual", "--as-cran"))
# 4 sprawdzam czy wszystkie linki w dokumentacji działają
# install.packages("urlchecker", repos = "https://r-lib.r-universe.dev")
urlchecker::url_check()
urlchecker::url_update()
# 5 aktualizuję NEWS i wersję pakietu
# 6 wysyłam pakiet do CRAN
devtools::submit_cran()
```

Bardziej złożone (i kompletne) podejście do przygotowania pakietu do CRAN jest opisane na stronie <https://github.com/ThinkR-open/prepare-for-cran>.

Po wywołaniu ostatniej z funkcji należy potwierdzić swój adres email, a następnie kliknąć w otrzymany mailowo link. W efekcie pakiet zostanie wysłany do CRAN, gdzie zostanie

sprawdzony przez automatyczne narzędzia.

W przypadku wysłania pakietu do CRAN po raz pierwszy, jest on sprawdzony manualnie przez jedną z osób w tzw. CRAN team. Taka osoba może albo zaakceptować pakiet, albo poprosić o poprawienie błędów i przesłanie pakietu ponownie.

Aktualizowanie pakietu przebiega w ten sam sposób – poprzez wywołanie `devtools::submit_cran()`.<sup>6</sup> Wtedy pakiet zostanie sprawdzony przez automatyczne narzędzia. W przypadku, gdy automatyczne testy zwrócą jakieś problemy (ostrzeżenia lub notatki), zostanie on również obejrzany przez CRAN team. Warto pamiętać, że nie należy aktualizować wersji CRAN pakietu częściej niż raz na miesiąc.

#### Komentarz

Możliwe jest sprawdzenie statusu pakietu przesłanego na CRAN na stronie <https://r-hub.github.io/cransays/articles/dashboard.html>.

Gdy pakiet jest już na CRAN, ale nie jest zgodny z nowymi wymaganiami, otrzyma się prośbę o jego naprawę, a w przypadku nie naniesienia poprawek, zostanie on usunięty z CRAN (*archived*).

#### Komentarz

Wysyłanie pakietu na CRAN nie jest jedynym sposobem na jego udostępnienie. W różnych sytuacjach możliwe jest też udostępnienie pakietu na innych repozytoriach, np. na GitHubie, GitLabie, Bitbucket, lub na własnym serwerze. Inne możliwości to np. użycie systemu [drat](#) lub [r-universe](#). Powyższe rozwiązania nie mają takich samych wymagań jak CRAN, ale wymagają więcej od potencjalnego użytkownika. Dodatkowo, [bioconductor](#) to repozytorium zawierające pakiety R związane z bioinformatyką.

---

<sup>6</sup>Możliwe jest też wysłanie pakietu do CRAN za pomocą strony <https://cran.r-project.org/submit.html>.

## 7 Kontrola wersji (3)

### 7.1 GitHub issues

Sprawy (ang. *issues*) to miejsce, gdzie twórcy mogą zapisywać swoje listy zadań dotyczące danej aplikacji, a użytkownicy mogą zgłaszać błędy czy propozycje ulepszeń. To miejsce można znaleźć na stronie głównej repozytorium, w zakładce *Issues*. Sprawy, po ich utworzeniu, mogą być przypisane do konkretnych osób, które są odpowiedzialne za ich rozwiązanie, a także mogą być oznaczone etykietami, które pomagają w ich kategoryzacji. Sprawy można również komentować, a po ich rozwiązaniu można je zamknąć. Każda sprawa otrzymuje unikalny numer, który może być użyty do odwołania się do niej w innych miejscach.

#### Komentarz

Jako osoba zgłaszająca sprawę należy podać jak najwięcej informacji pomagających odtworzyć problem. Najlepiej w takiej sytuacji jest przygotować tzw. powtarzalny przykład (ang. *reprex*), który zawiera minimalny kod potrzebny do odtworzenia problemu. Krótki opis działania powtarzalnych przykładów można znaleźć pod adresem <https://jakubnowosad.com/elp/debugging.html#reprex>.

### 7.2 Pull requests

Zapytania aktualizacyjne (ang. *pull requests*) to sposób na zaproponowanie zmian w repozytorium. Co warto podkreślić, takie zapytania można nanieść zarówno na swoje własne repozytorium, jak i repozytorium innych użytkowników.

Zapytanie aktualizacyjne we własnym repozytorium wiąże się zazwyczaj z (1) stworzeniem nowej gałęzi, (2) naniesieniem zmian, np. w kodzie, (3) zatwierdzeniem zmian, (4) przesłaniem zmian do zdalnego repozytorium, (5) otwarciem zapytania aktualizacyjnego do głównej gałęzi repozytorium. Następnie można takie zmiany zaakceptować lub odrzucić.<sup>1</sup>

Zapytanie aktualizacyjne przydaje się również w przypadku, gdy chcemy zaproponować zmiany w repozytorium innego użytkownika. Może to być zarówno dodanie nowej możliwości, naprawienie błędu w kodzie, czy nawet poprawienie literówki w dokumentacji. W takiej sytuacji często opiera się to o (1) stworzenie rozwidlenia (ang. *fork*), (2) pobranie

---

<sup>1</sup>Lub zignorować...

rozwidlenia jako lokalne repozytorium, (3) edycja lokalnego repozytorium, (4) zatwierdzenie zmian i wysłanie ich do zdalnego repozytorium (rozwidlenia), (5) zaproponowanie zapytania aktualizacyjnego.

## 7.3 Model pracy

Do tej pory omówiliśmy wiele aspektów pracy z systemem Git i serwisem GitHub. Teraz pora na to, aby zastanowić się, jak to wszystko może wyglądać w praktyce pracy nad projektem.

Istnieje szereg możliwych modeli pracy z systemem Git i serwisem GitHub. Ich wybór zależy od tego ile osób pracuje nad projektem, skali tego projektu (czy jest to projekt jednorazowy, czy też projekt, który będzie rozwijany przez dłuższy czas), czy projekt jest otwarty na zewnętrzne kontrybucje, czy też od preferencji czy zwyczajów osób pracujących nad projektem.

W przypadku pracy jednej osoby nad krótkim projektem, model pracy może być bardzo prosty. Wystarczy, że osoba ta będzie pracować na głównej gałęzi (ang. *main*) i będzie regularnie wysyłać lokalne zmiany do zdalnego repozytorium.

Taki model może jednak nie działać za dobrze w przypadku pracy wielu osób nad projektem. Wiele równoległych zmian wprowadzanych przez różnych twórców może prowadzić do konfliktów<sup>2</sup>, a także powoduje, że trudno jest zrozumieć i planować zmiany wprowadzane do projektu. Wraz ze wzrostem liczby osób pracujących nad projektem, a także wraz ze wzrostem skali projektu, warto zastanowić się nad bardziej złożonym modelem pracy.

Jednym z takich modeli jest model *Feature branch*. Polega on na tym, że każda potencjalna zmiana wprowadzana do projektu jest tworzona w osobnej gałęzi (ang. *branch*). Wszystkie gałęzie są tworzone na bazie głównej gałęzi (ang. *main*). Po zakończeniu pracy nad zmianą tworzone jest zapytanie aktualizacyjne do głównej gałęzi, a następnie (po recenzji i zatwierdzeniu zmian przez inną osobę) zmiany z tej gałęzi są łączone z główną gałęzią.

### Komentarz

Istnieją też różne inne modele pracy, np. [Gitflow](#) w którym główna gałąź zawiera stabilną wersję projektu, a nowe funkcjonalności są dodawane do gałęzi *develop*.

W przypadku stosowania modelu pracy *Feature branch* warto stosować systemy CI/CD, które pozwalają na automatyczne sprawdzenie czy nowe zmiany nie powodują błędów w kodzie.

---

<sup>2</sup>technicznych i nie tylko

## **i** ZADANIA

1. Dobierzcie się w dwuosobowe grupy: każda z osób będzie pracowała na repozytorium pakietu drugiej osoby.
2. Niech każda z osób otworzy sprawę w repozytorium pakietu drugiej osoby. Sprawa powinna zawierać powtarzalny przykład użycia pakietu.
3. Niech każda z osób otworzy zapytanie aktualizacyjne do repozytorium pakietu drugiej osoby poprzez stworzenie rozwidlenia.
4. Rolą osoby, która jest właścicielem repozytorium, jest zaakceptowanie zapytania aktualizacyjnego i zamknięcie sprawy.

## 7.4 Konflikty Git

Konflikt Git to sytuacja, w której Git nie jest w stanie samodzielnie rozwiązać konfliktu pomiędzy dwoma wersjami pliku/ów. Może to mieć miejsce w przypadku, gdy dwie osoby pracujące nad tym samym projektem wprowadziły zmiany w tym samym miejscu w pliku. Pierwsza z osób przesłała swoje zmiany do zdalnego repozytorium, a druga próbuje przesłać swoje zmiany do tego samego zdalnego repozytorium (Seksja 3.6).

Zawsze warto zacząć od sprawdzenia, które pliki są konfliktowe. Można to zrobić oglądając okno Git w RStudio lub używając polecenia `git status` w terminalu. Dalej należy otworzyć problematyczny plik i znaleźć miejsce konfliktu. Będzie ono oznaczone specjalnymi znacznikami `<<<<<<<, =====, >>>>>>>`.

```
<<<<<<< HEAD:plik.R
x = 1 + 2
=====
y = 1 + 2
>>>>>>> zmiana:plik.R
```

Pierwszy znacznik, `<<<<<<< HEAD:`, oznacza treść, która jest w lokalnym repozytorium. Drugi znacznik, `=====`, oznacza miejsce, w którym następuje rozdzielenie pomiędzy lokalnym repozytorium a zdalnym repozytorium. Po nim jest nowa treść, która jest w zdalnym repozytorium. Trzeci znacznik, `>>>>>>> zmiana:`, oznacza koniec konfliktu.

Takie konflikty (ang. *merge conflicts*) mogą być rozwiązywane na różne sposoby. Najczęściej w takiej sytuacji można ręcznie edytować plik, stworzyć spójną wersję, usunąć znaczniki konfliktu, a następnie zatwierdzić zmiany (`git add plik.R` oraz `git commit`).

**i** Komentarz

Używając komendy `git merge --abort` można anulować łączenie zmian i wrócić do poprzedniego stanu przed rozpoczęciem scalania.

Inna możliwość to [zaakceptowanie tylko przychodzących lub obecnych zmian](#).

**i** Komentarz

Czasem może się zdarzyć, że pracując nad projektem zapędzimy się w szereg problematycznych zmian, które nie pozwalają nam na proste zatwierdzenie zmian i wysłanie ich do zdalnego repozytorium. W takiej sytuacji popularnym podejściem jest zrobienie kopii naszych lokalnych zmian w innym folderze komputera, usunięcie lokalnego repozytorium, ponowne sklonowanie zdalnego repozytorium, a następnie wklejenie naszych zmian do sklonowanego (czystego) repozytorium.



Rysunek 7.1: <https://xkcd.com/1597/>

Więcej porad na temat rozwiązywania konfliktów można znaleźć pod łatwym do zapamiętania adresem <https://ohshitgit.com/>.

## 7.5 Inne kwestie związane z systemem Git i serwisem GitHub

### 7.5.1 Identyfikatory

Z Gitem czy GitHubem powiązanych jest wiele różnych identyfikatorów, które mogą być przydatne w różnych sytuacjach. Pierwsze z nich to grupa identyfikatorów nazywanych zbiorczo *ref*, np.:

- HEAD – wskaźnik na aktualną wersję; HEAD~1 – wskaźnik na poprzednią wersję (poprzednie zatwierdzenie zmian)
- Nazwa gałęzi, np. main – wskaźnik na główną gałąź
- Tag, np. v1.0.0 – wskaźnik na konkretną wersję projektu
- Identyfikator zatwierdzenia zmian, np. a1b2c3d – wskaźnik na konkretny commit<sup>3</sup>

Kolejnym identyfikatorem jest nazwa użytkownika lub nazwa organizacji, która jest właścicielem repozytorium: `github.com/<nazwa_użytkownika>` lub `github.com/<nazwa_organizacji>`.

Dalej po nazwie użytkownika lub organizacji podawana jest nazwa repozytorium: `github.com/<nazwa_użytkownika>/<nazwa_repozytorium>` lub `github.com/<nazwa_organizacji>/<nazwa_repozytorium>`.

Wewnątrz każdego repozytorium można tworzyć sprawy oraz zapytania aktualizacyjne. Każda sprawa i zapytanie aktualizacyjne ma swój unikalny identyfikator, który jest wyliczeniem: `github.com/<nazwa_użytkownika>/<nazwa_repozytorium>/issues/<numer_sprawy>` lub `github.com/<nazwa_użytkownika>/<nazwa_repozytorium>/pull/<numer_zapytania>`.<sup>4</sup> Takie identyfikatory pozwalają na odwoływanie się do konkretnych spraw i zapytań aktualizacyjnych w różnych miejscach za pomocą znaku # i numeru identyfikatora, np. #1 lub #2.

Mój komentarz dotyczy sprawy #1.

Do danej sprawy można odnieść się w tekście innej sprawy, w zapytaniu aktualizacyjnym, w komentarzu, w kodzie, w dokumentacji, a nawet w tekście zatwierdzenia zmian.

#### Komentarz

W tekście zatwierdzenia zmian czy zapytaniu aktualizacyjnym można nawet użyć jednego ze słów kluczowych, które automatycznie zamknie sprawę, takich jak *closes*, *fixes*, *resolves*

Naprawiono błąd, closes #1.

<sup>3</sup>To jest skrótowy identyfikator, pełny identyfikator to 40-znakowy ciąg znaków.

<sup>4</sup>Traktowane są one łącznie, więc przykładowo po sprawie numer jeden będzie zapytanie aktualizacyjne numer dwa.



## 7.5.2 Szablony

GitHub pozwala także na tworzenie, a następnie korzystanie z różnych szablonów. Możliwe przykładowo jest stworzenie szablonu całego repozytorium. Polega to tylko na zbudowaniu repozytorium z odpowiednimi plikami, które będą stanowić szablon, a następnie [wybranie w opcjach repozytorium](#). Następnie każde nowe repozytorium będzie można utworzyć na podstawie tego szablonu.

Możliwe jest również stworzenie [szablonu sprawy](#) (często stosowane do pokazania użytkownikom w jaki sposób zgłaszać błędy czy problemy) lub [zapytania aktualizacyjnego](#).

## 7.5.3 Ograniczenia wielkości plików

Głównym celem systemu Git jest przechowywanie kodu źródłowego, a nie dużych plików. GitHub pozwala na przechowywanie plików o maksymalnej wielkości 100 MB, ale zaleca, aby pliki były mniejsze niż 50 MB. Dodatkowo, wielokrotne nadpisywanie dużych plików w repozytorium może prowadzić do jego nadmiernego rozrostu.<sup>5</sup>

Co zrobić w przypadku, gdy nasz kod używa dużych plików, np. plików z danymi? Pierwsza strategia to kompresja takich plików oraz unikanie ich częstej aktualizacji. Drugą strategią jest przechowywanie takich plików w innym miejscu, np. na własnym serwerze lub w serwisie [Zenodo](#), a następnie w kodzie wewnątrz repozytorium umieszczenie jedynie odnośnika do takiego pliku. Kolejna strategia to użycie Git LFS (ang. *Large File Storage*), który pozwala na przechowywanie dużych plików w innym miejscu, a następnie w repozytorium umieszczenie jedynie odnośnika do takiego pliku. Więcej o tej ostatniej strategii można przeczytać pod adresem <https://docs.github.com/en/repositories/working-with-files/managing-large-files>.

## 7.5.4 GitHub Pages

GitHub, jak już wcześniej wspomnieliśmy, nie tylko pozwala na przechowywanie i dzielenie się kodem, ale również na wiele innych rzeczy, takich jak dzielenie się małymi danymi czy przechowywanie (statycznych) stron internetowych. Każde repozytorium może mieć przypisaną stronę internetową, która jest dostępna pod adresem `https://<nazwa_użytkownika>.github.io/<nazwa_repozytorium>` – [taka strona może być nawet prostym dokumentem HTML](#). Co więcej, możliwe jest również posiadanie głównej strony internetowej użytkownika, która jest dostępna pod adresem `https://<nazwa_użytkownika>.github.io`.<sup>6</sup> Taka strona jest tworzona w oparciu o repozytorium o nazwie `<nazwa_użytkownika>.github.io`. Więcej na temat tworzenia różnorodnych dokumentów HTML w oparciu o język R można znaleźć na stronie <https://quarto.org/>, a na temat tworzenia stron internetowych w serwisie GitHub można znaleźć na stronie [GitHub Pages](#).

---

<sup>5</sup>Możliwe jest [usunięcie dużych plików z historii zmian Git](#).

<sup>6</sup>Możliwe jest powiązanie jej z własną domeną.

## 7.6 Dodatkowe materiały

Z racji popularności (i złożoności) systemu Git istnieje ogromna liczba materiałów pomagających w jego nauce i zrozumieniu oraz wiele stron zawierających pytania i odpowiedzi dotyczące napotkanych problemów. W przypadku łączenia możliwości języka R z systemem Git warto poczytać materiały zawarte na stronie <https://happygitwithr.com/> (Bryan, the STAT 545 TAs, i Hester 2019) oraz rozdział [Software development practices](#) książki *R packages* (Wickham 2015). Do ogólnego wprowadzenia do systemu Git może posłużyć darmowa książka online [Pro Git](#) (Chacon 2014), której kilka pierwszych rozdziałów jest również dostępna w języku polskim czy też dokument [System kontroli wersji Git](#). Git jest również bardzo popularnym tematem na serwisie stackoverflow, gdzie można znaleźć [pytania i odpowiedzi na różnorodne tematy z nim związane](#). Więcej odnośników do materiałów związanych z systemem Git i serwisem GitHub można znaleźć na stronach [pomocy GitHub](#).

JULIA EVANS  
@b0rk

## git discussion bingo

WTF is detached HEAD state	I hate git	git's design is so elegant	I've used git for 10 years and I have no idea how it works	commits are immutable snapshots
just use magit	git is a directed acyclic graph	you have to understand the linux kernel dev workflow	mercurial is better	git is not github
subversion was so much worse	I only know 5 commands	a branch is just a pointer to a commit	the CLI is badly designed	you should just read Pro Git
rewriting history is bad	just spend 15 minutes learning git's internals	something about "porcelain"	merge sucks, only use rebase	rebase sucks, only use merge
I just do not care how git works	content addressed storage	subversion was better	something about Linus Torvalds	I just delete my git repo if I mess it up

Rysunek 7.2: <https://social.jvns.ca/@b0rk/111460966674032287>

## 8 Tworzenie pakietów R (4)

### 8.1 Zaawansowane dokumentowanie funkcji

Sekcje 2.5 oraz 4.2 omawiały plik DESCRIPTION oraz jego zawartość. Jedną z kwestii, która nie została tam poruszona jest dodanie do pakietu możliwości dokumentowania funkcji używając języka Markdown. W tym celu należy w pliku DESCRIPTION dodać linię Roxygen: `list(markdown = TRUE)`.

```
Package: mojpakiet
Title: Moje Funkcje Robiace Wszystko
Version: 0.0.1
Authors@R:
  person(given = "Imie",
         family = "Nazwisko",
         role = c("cre", "aut"),
         email = "imie.nazwisko@example.com")
Description: Tworzenie, przeliczanie i wyliczanie wszystkiego.
  Czasami nawet więcej.
License: CC0
Encoding: UTF-8
LazyData: true
RoxygenNote: 7.2.3
Roxygen: list(markdown = TRUE)
```

Teraz możliwe jest stosowanie znaczników Markdown w dokumentacji funkcji. Poniżej pokażę ich użycie oraz inne, wcześniej nie omówione, możliwości pakietu **roxygen2** (Sekcja 2.7).

```
#' Konwersja odleglosci
#'
#' @description Funkcja sluzaca do konwersji odleglosci z mil na kilometry
#'
#' @param mil wektor zawierajacy wartosci odleglosci w milach
#'
#' @details Funkcja `konwersja_odl()` konwertuje odleglosci z
#'   [mil](https://en.wikipedia.org/wiki/Mile) na kilometry używając wzoru
#'   przelicznika: 1 mila = 1.609344 km
```

```

#'
#' Ta funkcja jest:
#' * Niezwykle przydatna
#' * Bardzo prosta
#'
#' @return wektor numeryczny
#' @export
#'
#' @references Tobler, W. R. (1970). "A Computer Movie Simulating Urban Growth
#'           in the Detroit Region". Economic Geography. 46: 234-240
#'
#' @seealso [konwersja_temp()], [units::set_units()]
#'
#' @examples
#' konwersja_odl(75)
#' konwersja_odl(110)
#' konwersja_odl(0)
#' \dontrun{
#'   konwersja_odl(c(0, 75, 110))
#' }
konwersja_odl = function(mil){
  mil * 1.609344
}

```

Język Markdown ma kilka zastosowań w dokumentacji funkcji. Po pierwsze pozwala na stylizowanie tekstu, np. pogrubienie, pochylenie, itp. Przykładowo,

```
`konwersja_odl()`
```

zamieni się w `konwersja_odl()`. Po drugie, Markdown pozwala na dodanie linków do innych stron internetowych, które mogą być przydatne w kontekście funkcji. Używa się wtedy składni `[mil](https://en.wikipedia.org/wiki/Mile)`, gdzie `mil` to tekst, który będzie wyświetlany, a `https://en.wikipedia.org/wiki/Mile` to link do strony internetowej.

Markdown pozwala również na odnoszenie się do dokumentacji innych funkcji, zarówno wewnątrz pakietu, jak i zewnętrznych. Wewnętrzne funkcje odwołuje się za pomocą `[konwersja_temp()]`, gdzie `konwersja_temp()` to nazwa funkcji, do której chcemy się odwołać, a zewnętrzne za pomocą `[units::set_units()]`, gdzie `set_units()` to nazwa funkcji, do której chcemy się odwołać, a `units` to nazwa pakietu, w którym ta funkcja się znajduje. Kolejną możliwością jest proste wstawianie list używając znacznika `*`.<sup>1</sup>

Dokumentacja funkcji może zawierać także inne znaczniki, które nie zostały omówione w poprzednich sekcjach. Obejmuje to takie znaczniki jak:

---

<sup>1</sup>Markdown umożliwia także inne kwestie, takie jak wstawianie obrazków, tabel, itp.

- `@details`: dodatkowe informacje o funkcji, w tym jej zastosowanie, ograniczenia, specjalne przypadki, itp.
- `@seealso`: odnośniki do innych funkcji, które mogą być podobne lub powiązane z funkcją, której dokumentacja jest tworzona
- `@references`: odnośniki do literatury (np. artykułu naukowego), na podstawie której powstała funkcja

Znacznik `@examples` pozwala na dodanie przykładów użycia funkcji. Część z tych przykładów można powstrzymać przed uruchomieniem używając znacznika `\dontrun{}`. Może się to przydać w kwestii, np. gdy chcemy pokazać przykład wystąpienia błędu.

Więcej informacji na temat dokumentowania funkcji można znaleźć na stronie <https://r-pkgs.org/man.html#roxygen2-basics>.

## 8.2 Dodatkowe możliwości pakietu `usethis`

Pakiet **`usethis`** także oferuje inne możliwości, które nie zostały omówione w poprzednich sekcjach. Nie są one niezbędne do tworzenia pakietów, ale mogą być przydatne w niektórych przypadkach. Tutaj pokrótce omówię kilka z nich:

- `use_citation()`: dodaje [plik `inst/CITATION`](#) do pakietu, który zawiera informacje w jaki sposób cytować dany pakiet. Zazwyczaj warto dodać ten plik, gdy dany pakiet powiązany jest z publikacją naukową.
- `use_package_doc()`: tworzy on plik `R/{nazwapakietu}-package.R`, który jest ogólną dokumentacją pakietu. Można go później wywołać poprzez `?{nazwapakietu}`.
- `use_cran_badge()`: dodaje plaketkę (ang. *badge*) z odnośnikiem do strony pakietu na CRAN do pliku `README`. Możliwe jest także dodanie innych plaketek używając funkcji `use_badge()`, w której należy podać nazwę i odnośnik do plakietki.<sup>2</sup>

Dodatkowo, pakiet **`usethis`** pozwala na dodawanie informacji do pakietu z poziomu R, np. przy użyciu funkcji `use_author()` czy `use_version()`.

---

<sup>2</sup>Polecam obejrzeć repozytoria GitHub innych pakietów po inspiracje.

## **i ZADANIA**

1. Dodaj do swojego pakietu możliwość dokumentowania funkcji używając Markdown.
2. Dodaj do jednej z funkcji dodatkowe informacje używając znaczników Markdown: odnośniki do innych funkcji, odnośniki do literatury, listę, itp.
3. Dodaj do swojego pakietu plik `inst/CITATION` używając funkcji `use_citation()`.
4. Dodaj do swojego pakietu plik `R/{nazwapakietu}-package.R` używając funkcji `use_package_doc()`.
5. Dodaj do swojego pakietu plaketkę z odnośnikiem do strony pakietu na CRAN używając funkcji `use_cran_badge()`.
6. Zaktualizuj dokumentację używając funkcji `devtools::document()`, a następnie załaduj pakiet używając funkcji `devtools::load_all()`. Sprawdź jakie zaszły zmiany w dokumentacji.
7. Prześlij wszystkie zmiany do repozytorium zdalnego na GitHubie.

## **8.3 Inne kwestie związane z tworzeniem pakietów R**

Do tej pory omówiliśmy najważniejsze kwestie związane z tworzeniem pakietów R. Jednakże, istnieje wiele innych aspektów, które nie zostały tutaj poruszone, ale warto zdawać sobie z nich sprawę.

Jednym z nich jest plik `NAMESPACE`, który jest generowany automatycznie przez pakiet **roxygen2**. Plik ten zawiera informacje o funkcjach, które są eksportowane z pakietu, a także o funkcjach, które są importowane z innych pakietów. Tego pliku nie należy edytować ręcznie.

Innym aspektem jest plik `.Rbuildignore`, który zawiera informacje o plikach, które mają być ignorowane podczas budowania pakietu. Warto dodać do tego pliku pliki, które nie są niezbędne do działania pakietu, ale są powiązane z jego tworzeniem. Przykładowo, obejmuje to plik projektu RStudio `^{\text{nazwapakietu}}.Rproj`, pliki czy folder związane z CI/CD (np. `^\.github$`), czy folder z surowymi danymi i kodem (np. `^data-raw$`).

Kolejnym aspektem są zależności pakietów (ang. *dependencies*). Z jednej strony, nasz pakiet może wymagać innych pakietów do działania, a z drugiej natomiast, nasz pakiet może być wymagany przez inne pakiety. Obie te kwestie są złożone i decyzje o nich mają różne konsekwencje.

W przypadku zależności naszego pakietu od innych pakietów, możemy określić je w pliku `DESCRIPTION` w sekcji `Imports` lub `Suggests`. W efekcie, nasz pakiet staje się zależny od innych pakietów, które są wymienione w tych sekcjach. Zmiany w tych pakietach (i ich

zależnościach) mogą mieć wpływ na nasz pakiet. Pełne drzewo zależności pakietu można wyświetlić używając funkcji `pkg_deps_tree()`.

Istnieją różne podejścia do określania tego jakie pakiety powinny być wymagane przez nasz pakiet. Jednym z podejść jest używanie minimalnej liczby zależności, inne natomiast obejmuje stosowanie zależności co do autorów których mamy zaufanie. Jakikolwiek założenie jest przyjęte, nie warto przesadzać z liczbą zależności, gdyż może to mieć negatywny wpływ na czas budowania pakietu, a także na jego stabilność.

Nasz pakiet R może być również wymagany przez inne pakiety—taka sytuacja nazywana jest zależnością wsteczną (ang. *reverse dependency*). W tym przypadku, zmiany w naszym pakiecie mogą mieć wpływ na inne pakiety. Chcemy, aby wprowadzane przez nas zmiany nie powodowały szerokich konsekwencji w innych pakietach. Wykonanie sprawdzenia wpływu zmian na zależności wsteczne może być wykonane używając funkcji `usethis::use_revdep()`.

## 8.4 Dodatkowe materiały

W celu poznania i zrozumienia złożonych aspektów tworzenia pakietów R cennymi źródłami wiedzy może być książki [R packages](#) (Wickham 2015) oraz [rOpenSci Packages: Development, Maintenance, and Peer Review](#) (rOpenSci i in. 2019). Dodatkowo, w niektórych przypadkach pomocna może być oficjalna dokumentacja [Writing R Extensions](#) (R Core Team 2019). Niezastąpione jest także czytanie kodu źródłowego innych pakietów R na GitHubie.



# Bibliografia

- Bryan, Jenny, the STAT 545 TAs, i Jim Hester. 2019. *Happy Git and GitHub for the user*. Chacon, Scott. 2014. *Pro Git*. Apress.
- Chang, Winston, Javier Luraschi, i Timothy Mastny. 2019. *profvis: Interactive Visualizations for Profiling R Code*. <https://CRAN.R-project.org/package=profvis>.
- Eddelbuettel, Dirk, Romain Francois, JJ Allaire, Kevin Ushey, Qiang Kou, Nathan Russell, Douglas Bates, i John Chambers. 2020. *Rcpp: Seamless R and C++ Integration*.
- Ganz, Carl, Gábor Csárdi, Jim Hester, Molly Lewis, i Rachael Tatman. 2019. *available: Check if the Title of a Package is Available, Appropriate and Interesting*. <https://CRAN.R-project.org/package=available>.
- Gillespie, Colin, i Robin Lovelace. 2016. *Efficient R Programming: A Practical Guide to Smarter Programming*. " O'Reilly Media, Inc."
- Hester, Jim. 2020. *bench: High Precision Timing of R Expressions*. <https://CRAN.R-project.org/package=bench>.
- Hester, Jim, Gábor Csárdi, Hadley Wickham, Winston Chang, Martin Morgan, i Dan Tenenbaum. 2020. *remotes: R Package Installation from Remote Repositories, Including 'GitHub'*. <https://CRAN.R-project.org/package=remotes>.
- R Core Team. 2019. *Writing R Extensions*. R Foundation for Statistical Computing.
- rOpenSci, Brooke Anderson, Scott Chamberlain, Anna Krystalli, Lincoln Mullen, Karthik Ram, Noam Ross, Maëlle Salmon, i Melina Vidoni. 2019. „rOpenSci Packages: Development, Maintenance, and Peer Review”, styczeń. <https://doi.org/10.5281/zenodo.2554759>.
- Wickham, Hadley. 2014. *Advanced R*. Chapman and Hall/CRC.
- . 2015. *R Packages: Organize, Test, Document, and Share Your Code*. " O'Reilly Media, Inc."
- . 2020. *testthat: Unit Testing for R*. <https://CRAN.R-project.org/package=testthat>.
- Wickham, Hadley, i Jennifer Bryan. 2020. *usethis: Automate Package and Project Setup*. <https://CRAN.R-project.org/package=usethis>.
- Wickham, Hadley, i Jay Hesselberth. 2020. *pkgdown: Make Static HTML Documentation for a Package*. <https://CRAN.R-project.org/package=pkgdown>.

# A Profiling

Istnieją trzy podstawowe reguły optymalizacji kodu<sup>1</sup>:

1. Nie.
2. Jeszcze nie.
3. Profiluj przed optymalizowaniem.

Czym jest profilowanie i dlaczego powinno być wykonywane przed optymalizowaniem kodu? Profilowanie mierzy wydajność działania każdej linii kodu w celu sprawdzenia, która linia zabiera najwięcej czasu lub zasobów. Dzięki profilowaniu można określić fragmenty kodu, które można poprawić w celu zwiększenia czasu wykonywania skryptu czy funkcji.

Poniżej znajduje się zawartość pliku `R/moja_funkcja.R`. Jego działanie polega na stworzeniu wektora od 1 do 9999999 (obiekt `x`), wektora od 1 do 19999998 co 2 (obiekt `y`), połączenie tych wektorów do ramki danych (obiekt `df`), wyliczenie sumy wartości dla każdego wiersza (obiekt `z`), a na końcu wyliczenie średniej z obiektu `z`. Która z tych linii zabiera najwięcej czasu a która najmniej?

```
# plik R/moja_funkcja.R
x = 1:9999999
y = seq(1, 19999998, by = 2)
df = data.frame(x = x, y = y)
z = rowSums(df)
mean(z)
```

Profilowanie kodu R można wykonać używając funkcji `profvis()` z pakietu **profvis** (Chang, Luraschi, i Mastny 2019). Przyjmuje ona kod lub funkcję, która ma zostać profilowana.

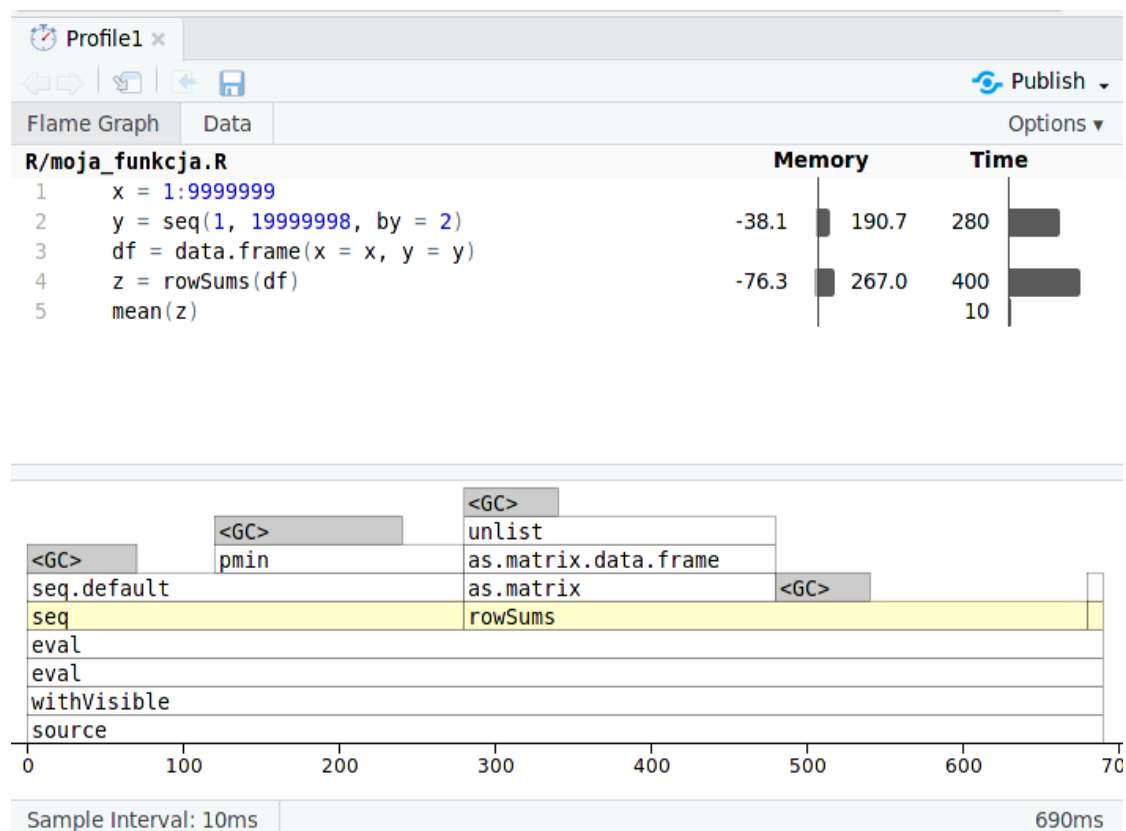
```
library(profvis)
profvis(source("R/moja_funkcja.R"))
```

W powyższym przypadku nastąpiło profilowanie kodu zawartego w skrypcie `R/moja_funkcja.R`. Efektem działania jest interaktywne podsumowanie pokazujące zużycie pamięci oraz czas poświęcony dla kolejnych linii kodu.

Czas wykonania tego przykładu wyniósł sumarycznie 690ms. Pierwsza linia tworząca obiekt `x` została wykonana bardzo szybko - poniżej mierzalnego progu. Stworzenie obiektu `y` w drugiej

---

<sup>1</sup><http://www.moscowcoffeereview.com/programming/the-3-rules-of-optimization/>



Rysunek A.1: Zrzut ekranu przedstawiający wynik działania funkcji `profvis()`

linii zajęło ok. 280ms. Trzecia linia została wykonana również w czasie poniżej mierzanego progu. Wynika to z kwestii, że tworzenie tam ramki danych nie powoduje wykonania nowych, złożonych obliczeń. Powstała ona jedynie poprzez przekazanie odpowiednich adresów w pamięci do obiektów  $x$  i  $y$ . Najbardziej czasochłonną okazała się linia czwarta. Wyliczenie sum wierszy i stworzenie obiektu  $z$  zabrało ok. 400ms. Ostatnia linia, wyliczająca średnią, zabrała ok. 10ms.

## B Benchmarking

Benchmarking oznacza określanie wydajności danej operacji czy funkcji. Wydajność może być określona na wiele różnych sposobów, w tym najprostszym jest czas wykonania pewnego kodu. Do określenia ile czasu zajmuje działanie operacji można użyć wbudowanej funkcji `system.time()`.

```
system.time(kod_do_wykonania)
```

Przykładowo, poniżej nastąpi sprawdzenie czasu jaki zajmie wyliczenie średniej wartości z sekwencji od 1 do 100000000.

```
system.time(mean(1:100000000))
```

```
      user  system elapsed  
0.535    0.000    0.536
```

W efekcie dostajemy trzy wartości - user, system i elapsed. Pierwsza z nich określa czas obliczenia po stronie użytkownika (sesji R), druga opisuje czas obliczenia po stronie systemu operacyjnego (np. otwieranie plików), a trzecia to sumaryczny czas wykonywania operacji.

Benchmarking jest często używany w sytuacji, gdy istnieje kilka funkcji służących do tego samego celu (np. w różnych pakietach) i chcemy znaleźć tę, która ma najwyższą wydajność. Jest on też stosowany, gdy sami napisaliśmy kilka implementacji rozwiązania tego samego problemu i chcemy sprawdzić, które z nich jest najszybsze.

Istnieje kilka możliwych implementacji pętli for pozwalających na przeliczanie wartości z mil łądowych na kilometry. Pierwsza z nich, tutaj zdefiniowana jako funkcja `mi_do_km1`, tworzy pusty wektor o długości 0, do którego następnie doklejane są kolejne przeliczone wartości.

```
mi_do_km1 = function(odl_mile){  
  odl_km = vector("list", length = 0)  
  for (i in seq_along(odl_mile)) {  
    odl_km = c(odl_km, odl_mile[[i]] * 1.609)  
  }  
  odl_km  
}
```

Druga, tutaj zdefiniowana jako funkcja `mi_do_km2`, tworzy pusty wektor o oczekiwanej długości wyniku. Następnie kolejne przeliczone wartości są wstawiane w odpowiednie miejsca wektora wynikowego.

```
mi_do_km2 = function(odl_mile){
  odl_km = vector("list", length = length(odl_mile))
  for (i in seq_along(odl_mile)) {
    odl_km[[i]] = odl_mile[[i]] * 1.609
  }
  odl_km
}
```

Dwie powyższe funkcje można porównać używając `system.time()`. Nie zawsze jednak to wystarczy - ta sama funkcja wykonana dwa razy może mieć różny czas obliczeń. Dodatkowo, oprócz czasu wykonywania funkcji może nas interesować zużycie zasobów, takich jak pamięć operacyjna. Do takiego celu powstała funkcja `mark()` z pakietu **bench** (Hester 2020), która wykonuje funkcję wiele razy przed zwróceniem wyniku.

Przyjmuje ona wywołania funkcji, które chcemy porównać. Poniżej nastąpi porównanie funkcji `mi_do_km1` i `mi_do_km2`, w przypadku gdy jako dane wejściowe zostanie podana lista z wartościami 142, 63, 121.

```
library(bench)
odl_mile = list(142, 63, 121)
wynik_1 = mark(
  mi_do_km1(odl_mile),
  mi_do_km2(odl_mile)
)
wynik_1
```

# A tibble: 2 x 6

	expression	min	median	`itr/sec`	mem_alloc	`gc/sec`
	<bch:expr>	<bch:tm>	<bch:tm>	<dbl>	<bch:byt>	<dbl>
1	mi_do_km1(odl_mile)	1.8us	2us	400133.	30.3KB	0
2	mi_do_km2(odl_mile)	1.4us	1.58us	601869.	27.8KB	0

Efektom porównania jest ramka danych, w której każdy wiersz oznacza inną porównywaną funkcję. Zawiera ona szereg charakterystyk, w tym:

- min - minimalny czas wykonania funkcji
- mean - średni czas wykonania funkcji
- median - mediana czasu wykonania funkcji
- max - maksymalny czas wykonania funkcji

- itr/sec - liczba wykonań funkcji na sekundę
- mem\_alloc - pamięć użyta przez wywołanie funkcji
- n\_itr - liczba powtórzeń wywołania funkcji

Wynik działania funkcji `mark()` pozwala na zauważenie, że na tym przykładzie funkcja `mi_do_km2` jest ok. 30% szybsza od `mi_do_km1`. Czasami możliwe jest, że jakaś funkcja działa relatywnie szybko na małych danych, ale dużo wolniej na większych danych wejściowych. Warto jest więc sprawdzić, jak będzie wyglądało nasze porównanie na większej liście, np. z wartościami od 0 do 10000 co 1.

```
odl_mile2 = as.list(0:10000)
wynik_2 = mark(
  mi_do_km1(odl_mile2),
  mi_do_km2(odl_mile2)
)
```

Warning: Some expressions had a GC in every iteration; so filtering is disabled.

```
wynik_2
```

```
# A tibble: 2 x 6
  expression      min    median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>    <bch:tm> <bch:tm>    <dbl>   <bch:byt>   <dbl>
1 mi_do_km1(odl_mile2) 683.93ms 683.93ms     1.46    382MB     11.7
2 mi_do_km2(odl_mile2)  1.11ms  1.15ms    793.     78.2KB     16.0
```

W tym przypadku różnica pomiędzy `mi_do_km1` a `mi_do_km2` staje się dużo większa. Funkcja `mi_do_km1` jest w stanie wykonać tylko 1.46 operacji na sekundę, przy aż 792.58 operacji na sekundę funkcji `mi_do_km2`. Dodatkowo, funkcja `mi_do_km1` potrzebowała aż kilka tysięcy (!) razy więcej pamięci operacyjnej niż `mi_do_km2`.

## C C++

R ma bezpośrednio wbudowane mechanizmy łączenia kodu R z kodem napisanym w językach C i Fortan. Dodatkowo, istnieje wiele pakietów, które pozwalają na łączenie R z innymi językami programowania, w tym C++.<sup>1</sup> C++ jest jednym z najczęściej używanych kompilowanych języków programowania. Jest to spowodowane kilkoma zaletami tego języka, w tym jego wysoką wydajnością, niezależnością od konkretnej platformy systemowej, czy uniwersalnością.

Język C++ posiada zarówno wiele podobnych do R konstrukcji i koncepcji, ale też różni się w pewnych kluczowych koncepcjach. Najważniejsze cechy C++, które wyróżniają go od R i które warto znać na początku:

- Jest językiem kompilowanym
- Pozwala na używanie tylko = jako operatora przypisania
- Zakłada głównie statyczną kontrolę typów
- Posiada typ skalarny
- Domyślnie nie używa wektoryzacji
- Większość linii kodu należy kończyć znakiem średnika ;
- Konieczne jest zwracanie wartości używając return

Przykładowy kod R do przeliczania temperatury ze stopni Fahrenheita na Celsjusza wygląda w poniższy sposób:

```
konwersja_temp = function(temperatura_f){  
  (temperatura_f - 32) / 1.8  
}
```

To samo obliczenie wykonane w języku C++ może wyglądać w ten sposób:

```
double konwersja_temp_cpp(double temperatura_f){  
  double temperatura_c = (temperatura_f - 32) / 1.8;  
  return temperatura_c;  
}
```

Odnosząc się do punktów z wcześniej wymienionej listy:

---

<sup>1</sup>Inne możliwe połączenia R to z językami takimi jak [Rust](#), [JavaScript](#), czy [Java](#).



- Jest językiem kompilowanym - gdybyśmy chcieli użyć powyższą funkcję jako program C++ musielibyśmy stworzyć kolejną funkcję `main()`, a następnie skompilować kod. Nie jest możliwe wykonywanie tego kodu linia po linii
- Pozwala na używanie tylko `=` jako operatora przypisania - nie możemy w nim użyć operatora `<-` czy `->`
- Zakłada głównie statyczną kontrolę typów - w powyższym przykładzie musieliśmy zadeklarować, że nasza funkcja `konwersja_temp_cpp`, nasz argument `temperatura_f` oraz zmienna `temperatura_c` będzie typu `double`. Zrobiliśmy to poprzez dodanie nazwy typu przed nazwą funkcji/argumentu/zmiennej. Co ważne, w tym języku też typy (zazwyczaj) nie są automatycznie konwertowane do innych typów jak ma to miejsce w R.
- Posiada typ skalarny - `double` może przechowywać tylko jedną wartość.
- Domyślnie nie używa wektoryzacji - powyższa funkcja `konwersja_temp_cpp()` zwróci błąd `Expecting a single value` w przypadku podania wektora numerycznego jako obiekt wejściowy. Aby użyć wektor wartości na wejściu konieczne jest napisanie pętli lub użycie innych podobnych konstrukcji.
- Większość linii kodu należy kończyć znakiem średnika `;`. Nie dotyczy to linii definiujących powstanie funkcji, rozpoczynających i kończących pętle czy wyrażenia warunkowe
- Konieczne jest zwracanie wartości używając `return`. W R użycie funkcji `return()` było opcjonalne.

Obecnie ponad dwa tysiące pakietów R łączy się z językiem C++ używając pakietu **Rcpp** (Eddelbuettel i in. 2020). Dodanie języka C++ do pakietu R często ma na celu przyspieszenie pewnych wymagających obliczeniowo zadań lub połączenie R z istniejącymi zewnętrznymi bibliotekami napisanymi w C++.

```
library(Rcpp)
```

Pakiet **Rcpp** pozwala na zarówno wywoływanie kodu C++ wewnątrz skryptów R, jak używając zewnętrznych plików o rozszerzeniu `.cpp`.

Ta część materiałów ma na celu pokazanie pełnych podstaw łączenia R z C++. Więcej informacji na ten temat można znaleźć na stronie <http://www.rcpp.org/>, w rozdziale [Rewriting R code in C++](#) książki Wickham (2014), sekcji [Rcpp](#) książki Gillespie i Lovelace (2016), oraz na stronie [Unofficial Rcpp API Documentation](#).

## C.1 Wywoływanie kodu C++ wewnątrz skryptu R

W przypadku krótkich fragmentów kodu C++ możliwe jest umieszczenie ich wewnątrz skryptu R jako obiekt tekstowy. Poniżej stworzono nowy obiekt `rcpp_fun1`, który zawiera wcześniejszą funkcję C++.

```
rcpp_fun1 = "
double konwersja_temp_cpp(double temperatura_f){
  double temperatura_c = (temperatura_f - 32) / 1.8;
  return temperatura_c;
}
"
```

W kolejnym kroku konieczne jest skompilowanie powyższego kodu i stworzenie połączenia pomiędzy nim a R za pomocą funkcji `cppFunction()`.

```
cppFunction(rcpp_fun1)
```

Od tego momentu możliwe jest korzystanie z funkcji `konwersja_temp_cpp()`. Możemy sprawdzić jej działanie poprzez podanie wybranej przez nas wartości, na przykład 75.

```
konwersja_temp_cpp(75)
```

```
[1] 23.88889
```

Warto jednak nadal pamiętać, że powyższa funkcja nie jest zvektoryzowana - możliwe jest podanie w niej tylko obiektu o długości 1. W przypadku zadeklarowania dłuższego obiektu wejściowego otrzymamy błąd:

```
konwersja_temp_cpp(c(0, 75, 110))
```

```
Error in eval(expr, envir, enclos): Expecting a single value: [extent=3].
```

Poniżej znajduje się funkcja `mile_na_km()`, która przyjmuje i zwraca obiekt o klasie `list` i zamienia wartości elementów tej listy z mil lądowych na kilometry.

```
mile_na_km = function(odl_mile) {
  odl_km = vector("list", length = length(odl_mile))
  for (i in seq_along(odl_mile)) {
    odl_km[[i]] = odl_mile[[i]] * 1.609
  }
  odl_km
}
```

Ta sama funkcja w języku C++ może wyglądać w ten sposób:

```

List mile_na_km_cpp(List odl_mile){
  int odl_mile_len = odl_mile.size();
  List result(odl_mile_len);
  for (int i = 0; i < odl_mile_len; i++){
    result[i] = odl_mile[i] * 1.609;
  }
  return result;
}

```

Zawiera ona szereg różnic od kodu R. Oprócz definicji typów, używania średnika i operatora return, widać tutaj także inną metodę wywołania funkcji oraz inny sposób definiowania pętli for.

Zadaniem linii `int odl_mile_len = odl_mile.size();` jest stworzenie nowej zmiennej skalarnej (`odl_mile_len`) o typie `integer (int)`. Ta nowa zmienna jest wynikiem działania funkcji `size()`, która jest odpowiednikiem używanej w R `length()`. W przypadku używania R wywołanie funkcji ma jednak postać, w której podajemy nazwę funkcji, a następnie w nawiasie okrągłym obiekt wejściowy. C++ pozwala też na inny sposób wywoływania funkcji - używając wbudowanych metod. Odbywa się to poprzez podanie nazwy obiektu (`odl_mile`), a następnie po kropce (`.`) podania nazwy funkcji (`size()`).

W C++ pętle można definiować używając poniższej składni:

```

for (inicjalizacja zmiennej; warunek zakończenia; aktualizacja zmiennej) {
  // Kod do wykonania
}

```

Po pierwsze należy przekazać w miejscu `inicjalizacja zmiennej` stworzenie zmiennej, na podstawie której będzie oparta pętla. `int i = 0` oznacza, że tworzymy zmienną o typie `integer i`, która przyjmuje wartość 0. Jest to spowodowane ważną różnicą między C++ a R - w tym pierwszym języku liczenie rozpoczynamy od 0. Przykładowo w C++, `a[0]` pozwoli na wybranie pierwszego elementu z wektora `a`. Drugim elementem jest warunek `trwania`, czyli określenie do kiedy pętla trwa. `i < odl_mile_len` oznacza, że pętla będzie działała tak długo aż `i` będzie mniejsze niż `odl_mile_len`. Ostatni element, `aktualizacja zmiennej`, mówi co ma się stać ze stworzoną zmienną po każdym przebiegu pętli. `i++` to skrót w języku C++ mówiący, że z każdą pętlą wartość `i` będzie rosła o 1. Jest to odpowiednik kodu `i = i + 1`.

W powyższej składni też widać sposób definiowania komentarzy w języku C++, gdzie używa się operatora `//`.

Zainicjujmy funkcję C++ `mile_na_km_cpp()` używając `cppFunction()`.

```

rcpp_fun2 = "List mile_na_km_cpp(List odl_mile){
  int odl_mile_len = odl_mile.size();
  List result(odl_mile_len);

```

```

    for (int i = 0; i < odl_mile_len; i++){
        result[i] = odl_mile[i] * 1.609;
    }
    return result;
}"
cppFunction(rcpp_fun2)

```

Możemy sprawdzić jej działanie na przykładowym wektorze `odl_mile` używając kodu poniżej.

```

odl_mile = list(142, 63, 121)
mile_na_km_cpp(odl_mile)

```

```

[[1]]
[1] 228.478

```

```

[[2]]
[1] 101.367

```

```

[[3]]
[1] 194.689

```

Dodatkowo warto porównać prędkość rozwiązania w R z C++ używając listy o długości 10001 i funkcji `mark()` z pakietu **bench**.

```

odl_mile2 = as.list(0:10000)
wynik = bench::mark(
  mile_na_km(odl_mile2),
  mile_na_km_cpp(odl_mile2)
)
wynik

```

```

# A tibble: 2 x 6
  expression          min    median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>      <bch:tm> <bch:tm>    <dbl>   <bch:byt>   <dbl>
1 mile_na_km(odl_mile2)  1.07ms  1.18ms     823.    112.1KB     15.0
2 mile_na_km_cpp(odl_mile2) 438.1us 467.52us    2081.     84.8KB     46.9

```

Mimo otrzymania tego samego wyniku, czas wykonania funkcji napisanej w C++ był około 2.53 raza mniejszy.

## C.2 Wywoływanie kodu z plików .cpp

Powyższy przykład sprawdza się w przypadku małych fragmentów kodu C++. W momencie jednak, gdy kod staje się bardziej złożony, znacznie lepsze jest umieszczenie go w oddzielnym pliku o rozszerzeniu .cpp. Taki plik może też być umieszczony wewnątrz pakietu R.

Używanie kod C++ z pliku z poziomu R wymaga jednak pewnych dodatkowych działań. Konieczne jest dodane do tego pliku kilku linii nagłówków, które umożliwią interakcję pomiędzy C++ a R.

Pierwsza z nich ma na celu umożliwienie dostępu do funkcji z pakietu **Rcpp**, poprzez podanie nazwy pliku Rcpp.h w linii #include.

```
#include <Rcpp.h>
```

Dalej, opcjonalnie można dodać również linię using namespace Rcpp;. W przeciwnym razie każde wywołanie funkcji z pakietu **Rcpp** (np., List) musielibyśmy poprzedzać Rcpp:: (np., Rcpp::List).

```
using namespace Rcpp;
```

Ostatnią kwestią przed dodaniem kodu funkcji C++ jest zdecydowanie czy konkretną funkcję chcemy udostępnić i używać w R. Gdybyśmy nie dodali poniższego kodu, funkcja nie byłaby widoczna z poziomu R.

```
// [[Rcpp::export]]
```

Kompletny kod funkcji można zobaczyć poniżej.

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
List mile_na_km_cpp(List odl_mile){
    int odl_mile_len = odl_mile.size();
    List result(odl_mile_len);
    for (int i = 0; i < odl_mile_len; i++){
        result[i] = odl_mile[i] * 1.609;
    }
    return(result);
}
```

Można go zapisać do pliku .cpp, np. mile\_na\_km\_cpp.cpp, a następnie skompilować i udostępnić dla R z użyciem funkcji sourceCpp().

```
sourceCpp("mile_na_km_cpp.cpp")
```

Teraz też możliwe jest jego sprawdzenie na przykładowym obiekcie:

```
odl_mile = list(142, 63, 121)
mile_na_km_cpp(odl_mile)
```

```
[[1]]
[1] 228.478
```

```
[[2]]
[1] 101.367
```

```
[[3]]
[1] 194.689
```

### C.3 C++ w pakietach R

Używanie kodu C++ wewnątrz pakietu R wymaga dodania kilku dodatkowych elementów. Po pierwsze należy dodać zależności do pakietu **Rcpp** w pliku DESCRIPTION:

```
Imports:
  Rcpp
LinkingTo:
  Rcpp
```

Najprościej to zrobić używając funkcji `usethis::use_rcpp()`.

Dalej należy przenieść kod C++ znajdujący się w pliku `.cpp` do folderu `src/` wewnątrz pakietu oraz uruchomić funkcję `Rcpp::compileAttributes()`, która wyeksportuje funkcje C++ do R. Teraz możliwe jest wywołanie stworzonej funkcji C++ wewnątrz funkcji w pakiecie R.

Używanie kodu C++ w pakietach R może również wymagać bardziej zaawansowanych działań, np. dodania pliku nagłówkowego `.h` lub edycji pliku `Makevars`. Wykracza to jednak daleko poza zakres tego szkolenia. Więcej o tworzeniu pakietów R używających C++ można znaleźć pod adresem <https://cran.rstudio.com/web/packages/Rcpp/vignettes/Rcpp-package.pdf>.