Elementarz programisty

wstęp do pogramowania używając R

Jakub Nowosad

Elementarz programisty

Wstęp do programowania używając R

Jakub Nowosad

2019-06-01

Wydanie pierwsze, Poznań 2019
Space A

Spis treści

O książce

Elementarz programisty: Wstęp do programowania używając R ma na celu wprowadzenie do podstaw działania w języku R. W pierszej części, "Podstawy", opisuje ona w jaki sposób wykonywać proste operacje w R, czym są obiekty oraz jak tworzyć funkcje, które je przetwarzają. Ta część zawiera także omówienie podstawowych narzędzi pozwalających na sterowanie przepływem informacji, takich jak wyrażenia warunkowe i pętle, metod działania na tekście, oraz sposobów wczytywania i zapisywania plików w różnych formatach. Druga część, "Narzędzia", buduje na wiedzy zdobytej w części pierwszej i rozszerza ją. Zawiera ona informacje na temat tworzenia funkcji przyjaznych użytkownikom oraz to w jaki sposób tworzyć odpowiednie komunikaty błędów, ostrzeżeń czy wiadomości. W tej części następuje też prezentacja metod analizy kodu, takich jak testy jednostkowe, benchmarking i profiling. Oprócz informacji ściśle powiązanych z R, **Elementarz programisty** ma także rozdział poświęcony systemom kontroli wersji - uniweralnym narzędziom używanym przez programistów różnych języków. Część "Narzędzia" kończy rozdział integrujący wiedzę z całej książki w postaci omówienia kolejnych kroków dotyczących tworzenia pakiektów R. **Elementarz programisty** zawiera też szereg praktycznych porad oraz wiele odnośników do dodatkowych materiałów, takich jak książki, blogi, kursy, czy strony internetowe. Wszystkie rozdziały w tej książce dodatkowo zawierają zadania, które pozwalają na sprawdzenie i utrwalenie wiedzy. Książka ta jest przeznaczona zarówno dla osób bez znajomości języków programowania, jak też dla osób, które znają inne języki programowania, ale są zainteresowane poznaniem języka R. Wiedza uzyskana poprzez używanie tej książki daje podstawy do wykorzystywania R do różnorodnych celów, od analizy danych i opracowań statystyczych, poprzez tworzenie wykresów i wizualizacji, kończąc na aplikacjach specyficznych dla danej dziedziny.

Aktualna wersja książki znajduje się pod adresem https://nowosad.github.io/elp/. Jeżeli używasz tej książki, zacytuj ją jako:

- Nowosad, J., (2019). Elementarz programisty: wstęp do programowania używając R. Poznań: Space A. Online: https://nowosad.github.io/elp/

Zachęcam również do zgłaszania wszelkich uwag, błędów, pomysłów oraz komentarzy na stronie https://github.com/nowosad/elp/issues.

Wymagania wstępne

Do odtworzenia przykładów oraz do wykonania zadań zawartych w tej książce 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 rodziałach użyte zostanie zintegrowane środowisko programistyczne **RStudio**. Można je zainstalować korzystając ze strony https://www.rstudio.com/products/rstudio/download/#download.

Aspekty dotyczące kontroli wersji zostaną omówione używając oprogramowania **Git**. Zalecanym sposobem installacji 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
```

Styl książki

W całej książce stosowana jest konwencja, w której `fun()` oznacza funkcje, `obi` oznacza nazwy obiektów, nazwy zmiennych oraz argumentów funkcji, a `sci/` oznacza ścieżki do plików. Wszystkie pakiety użyte w tej książce oznaczane są pogrubioną czcionką - **pak**.

Tekst na szarym tle przedstawia blok kodu. Może on zawierać komentarze (rozpoczynające się of znaku `#`), kod oraz wynik jego użycia (rozpoczynające się od znaków `#>`).

```
# komentarz
kod
#> wynik użycia kodu
```

Dodatkowo, ikona kompasu na szarym tle przedstawia dodatkowe informacje, alternatywne sposoby użycia funkcji, czy też wskazówki.

Tutaj może znaleźć się dodatkowa informacja, alternatywny sposób użycia funkcji, czy też wskazówka.

Podziękowania

Książka została stworzona w R (R Core Team, 2019a) z wykorzystaniem pakietów **bookdown** (Xie, 2018), **rmarkdown** (Allaire et al., 2019), **knitr** (Xie, 2019) oraz programu Pandoc[1].

Rysunek na okładce książki "Great-billed Heron (Ardea rectirostris)" został stworzony przez Elizabeth Gould do książki Birds of Australia Johna Goulda i został udostępniony na licencji CC0 1.0.

Ikony użyte w tej książce zostały stworzone przez Freepik z www.flaticon.com na licencji CC 3.0 BY.

[1] http://pandoc.org/

1. Wprowadzenie

Żyjemy obecnie w epoce trzeciej rewolucji przemysłowej[1], zwanej inaczej rewolucją cyfrową. Jest ona powiązana z przejściem z technologii mechanicznych i analogowych na technologie elektroniczne i cyfrowe. W tej epoce nastąpiło stworzenie i rozpowszechnieniem się komputerów, co w efekcie spowodowało szerokie zmiany społeczno-ekonomiczne. Wiele z tych zmian jest pozytywnych, ale istnieją również zmiany negatywne, bądź też takie które trudno jednoznacznie ocenić. Przykładowo, wyraźną korzyścią społeczną jest znacznie ułatwiony dostęp do informacji. Jednocześnie taki dostęp powoduje sytuację określaną jako przeciążenie informacją (ang. *information overload*), w której występuje zbyt wielka ilość informacji aby podjąć właściwą decyzję lub zrozumieć sens danego tematu.

Rozwój technologiczny spowodował też transformację produkcji przemysłowej i zmiany gospodarcze. Firmy zajmujące się technologiami informacyjnymi, tj. Microsoft, Apple, czy Google, są obecnie jednymi z najbardziej dochodowych przedsiębiorstw, a twórca platformy Amazon, Jeff Bezos, jest najbogatszym człowiekiem świata[2]. Wiele z tych technologii nie byłoby możliwych bez programowania. Programowanie, w znacznym uproszczeniu, to proces tworzenia serii instrukcji, które informują komputer jak wykonać pewne zadanie. Ta seria instrukcji jest zazwyczaj zapisywana na komputerze w postaci tekstu w wybranym języku programowania. Co w takim razie powoduje, że programowanie ma tak istotny wpływ na wiele elementów codziennego życia?

Programowanie cechuje kilka unikatowych możliwości. Po pierwsze, programowanie i jego efekty można w prosty sposób powielać niemal w nieskończoność. Wcześniej stworzenie pewnego towaru opierało się o ograniczone zasoby, np. ziemia czy surowce naturalne. Nie możliwe było wykucie zbroi raz, a następnie natychmiastowe powielenie jej wiele razy i sprzedanie jej wielu kopii. We współczesnym świecie, jedna aplikacja może być sprzedana (lub rozpowszechniona) wiele razy, a często większy nacisk kładzie się na rozbudowę i ulepszanie istniejących popularnych aplikacji niż tworzenie nowych[3]. Ułatwia to też budowę nowych rozwiązań na podsta-

[1] Niektórzy wydzielają już nawet obecny czas jako czwartą rewolucję przemysłową - https://en.wikipedia.org/wiki/Industry_4.0.

[2] https://en.wikipedia.org/wiki/The_World%27s_Billionaires#2018

[3] Efektem tego jest też coraz większa popularność modeli subskrypcyjnych - https://en.wikipedia.org/wiki/Subscription_business_model.

wie już istniejących[4]. Współcześnie programowanie pozwala na wykonywanie trylionów (10^{18}) operacji arytmetycznych na sekundę[5]. Pozwala to na znaczne zwiększenie wydajności dostępnych rozwiązań, otwiera możliwość praktycznego wykorzystania istniejących idei, lub też tworzenia nowych pomysłów. Inną cechą programowania jest też jego prosta możliwość automatyzacji powtarzanych czynnośći oraz ułatwiona powtarzalność (ang. *reproducibility*) Posiadając kod źródłowy danego oprogramowania lub skrypt wykonujący analizę danych, możliwe jest odtworzenie tego wyniku przez inną osobę na drugim końcu świata, lub też przez siebie samego po paru miesiącach. Ostatnią cechą programowania jest jego uniwersalność. Jest ono wykorzystywane w transporcie, przemyśle, nauce, rozrywce i wielu innych strefach życia. W efekcie zrozumienie i znajomość języków programowania jest cenną umiejętnością we współczesnym świecie.

1.1. Mity programistyczne

Programowanie komputerowe ma obecnie już długą historię - pierwszy język programowania Plankalkül powstał w latach 1943-1945[6]. Fortran, stworzony w roku 1957, jest nadal używany współcześnie do wielu celów, między innymi wymagających dużej wydajności obliczeń hydrologicznych, prognozowania pogody czy modelowania klimatu. Programowanie ewoluowało i nadal ewoluuje wraz z rozwojem dostępności i możliwości komputerów. Pojawiły się nowe pradygmaty programowania oraz wiele nowych języków. W tym samym czasie narosło również wiele mitów dotyczących programowania[7].

Jednym z mitów jest to, że programowanie polega tylko siedzeniu przed ekranem komputera i wpisywaniu do niego kolejnych linii kodu. Jest to oczywiście istotna część pracy programistycznej, ale prawdopodobnie nie jest ona nawet dominująca w przeciętym dniu programisty. Wcześniej konieczne jest zastanowienie się jaki problem rozwiązujemy oraz zaprojektowanie możliwego rozwiązania tego problemu. Stworzony kod może okazać się być nieprzystępny dla użytkownika, słabo zoptymalizowany, lub nawet błędny. Dlatego też innym ważnym elementem jest testowanie kodu w celu wyłapania potencjalnych problemów. Innym aspektem programowania jest tworzenie dokumentacji. Żaden program nie może zachęcić do siebie użytkowników, jeżeli nie będą oni w stanie zrozumieć jak on działa. Dokumentacja jest też cenna dla twórców programu, szczególnie kiedy konieczne jest użycie czy modyfikacja programu kilka miesięcy po jego ostatnim

[4] https://en.wikipedia.org/wiki/Standing_on_the_shoulders_of_giants

[5] Przecięty człowiek jest w stanie wykonać około pół operacji na sekundę - https://en.wikipedia.org/wiki/Computer_performance_by_orders_of_magnitude.

[6] https://en.wikipedia.org/wiki/Plankalk%C3%BCl

[7] Zobacz porównanie oczekiwań i rzeczywistej pracy programisty na https://www.youtube.com/watch?v=HluANRwPyNo.

użyciu. Programy komputerowe są też zazwyczaj w dużej sieci powiązań z już istniejącymi bibliotekami czy oprogramowaniem. Zmiana w tych bibliotekach czy oprogramowaniu może skutkować nie zawsze oczekiwanymi zmianami w stworzonym programie. Częścią programowania jest też utrzymywanie istniejącego kodu źródłowego oraz jego ulepszanie. Programiści do swojej pracy wykorzystują też odpowiednie wspierające ich narzędzia, takie jak edytory kodu źródłowego, debugery, zintegrowane środowiska programistyczne czy systemy kontroli wersji.

Mitem również jest przekonanie, że programowanie to męskie zajęcie. Bierze się ono z obecnej na rynku pracy struktury, w której około 75% programistów to mężczyźni a tylko 25% to kobiety. Ta struktura jednak nie jest odzwierciedleniem jakichś wrodzonych umiejętności. Za pierwszego programistę często uważa się Adę Lovelace, angielskiego matematyka i poetkę[8]. To ona w 1843 opublikowała pierwszy program komputerowy. Jej algorytm do obliczenia liczb Bernoulliego nie został jednak przetestowany, ponieważ urządzenie do tych obliczeń (zwane maszyną analityczną[9]) nie zostało skonstruowane. Ponad wiek później, gdy istniały już techniczne możliwości tworzenia komputerów, programowanie było uważane za kobiecy zawód[10] (Rycina 1.1). Z uwagi na szereg czynników społecznych i historycznych[11], w latach 1970 nastąpiło odwrócenie proporcji w tym zawodzie. Obecnie podejmowanych jest szereg inicjatyw, które mają na celu zachęcić kobiety do programowania. Wsród nich można wymienić działania organizacji R-Ladies[12], PyLadies[13], czy girls.js[14]. Mit programisty mężczyzny jest też powiązany z wymienionym kilka akapitów niżej mitem samotnego programisty.

Kolejny jest mit wielkiego produktu. Oznacza on, że po nauczeniu się podstaw danego języka programowanie, jest się od razu w stanie stworzyć bardzo złożony program, np. nowy system operacyjny, skomplikowaną aplikację na telefon, czy grę komputerową. W rzeczywistości takie produkty opierają się o tysiące godzin pracy wielu programistów. Dodatkowo, nie są one tworzone od podstaw, ale używając szeregu dostępnych narzędzi, bibliotek i innych rozwiązań. Celem pisania kodu, więc nie powinno być stworzenie od zera bardzo złożonej aplikacji, lecz odpowiednie użycie istniejących rozwiązań. Jednocześnie pisanie złożonego oprogramowania wymaga uzyskania niezbędnego doświadczenia. Mit wielkiego produktu wiąże się również z wymienionym w kolejnym akapicie mitem samotnego programisty.

[8] https://en.wikipedia.org/wiki/Ada_Lovelace
[9] https://en.wikipedia.org/wiki/Analytical_Engine
[10] https://www.history.com/news/coding-used-to-be-a-womans-job-so-it-was-paid-less-and-undervalued
[11] http://www.smbc-comics.com/?id=1883
[12] https://rladies.org/
[13] https://www.pyladies.com/
[14] https://girlsjs.pl/

Rysunek 1.1.: Margaret Hamilton stojąca w 1969 roku obok wydruków oprogramowania, które on i jej zespół stworzył na potrzeby misji Apollo. Źródło: https://commons.wikimedia.org/wiki/File:Margaret_Hamilton_-_restoration.jpg

W popkulturze osoba, która potrafi programować spędza czas samotnie, gwałtownie wpisując kolejne linie kodu do komputera w ciemnym pokoju. W rzeczywistości jednak większość profesjonalnych programistów pracuje w zespołach, których członkowie pracują nad różnymi aspektami tego samego problemu. Pisanie programów często wymaga współpracy różnych osób, dlatego też umiejętność pracy w grupie jest coraz istotniejsza. Warto dodać, że współpraca nad pisaniem programów nie musi odbywać się w jednym pokoju czy budynku. Ze względu na charakter takiej pracy i możliwości technologiczne, wiele formalnych i nieformalnych grup pracuje zdalnie nad projektami. Wiele przykładów takich zachowań można znaleźć przyglądając się otwartemu oprogramowaniu (ang. *open-source software*) na platformie GitHub (np. https://github.com/trending/r).

W poprzednim akapicie celowo użyłem stwierdzenia "osoba, która potrafi programować" zamiast "programista". Jest to kolejny powszechny mit, że każda osoba która potrafi stworzyć program musi od razu zostać pełnoetatowym programistą. Pisanie programów jest narzędziem, które ma wspomóc twórcę w pewnym celu. Jednym z celów może być zostanie profesjonalnym deweloperem stron internetowych, aplikacji mobilnych, gier komputerowych, itd. Nie jest to jednak jedyny cel - programowanie może być, na przykład przydatnym narzędziem w analizie danych[15]. Umiejętności programistyczne są wykorzystywane przez ekonomistów, biologów, geografów i osób z wielu innych dziedzin. Dodatkowo, podstawowe aspekty programowania są bardzo cenne w zawodach, w których ważna jest częsta współpraca z programistami.

Kolejny mitem jest mit programisty geniusza. W tym micie programują tylko osoby, która ma nadludzką pamięć oraz wyróżniającą wiedzę matematyczną. Oczywiście, takie cechy przydają się w programowaniu, ale nie są one wymagane do programowania. W programowaniu częściej od dobrej pamięci przydaje się umiejętność szybkiego znalezienia rozwiązania czy odpowiedzi na problem w internecie. Programista nie musi znać na pamięć setek różnych poleceń i funkcji, ważne że umie je zidentyfikować. Natomiast zamiast głębokiej wiedzy matematycznej do większości zadań programistycznych wystarczy podstawowa znajomość algebry. Z tym mitem wiąże się też inna kwestia - założenia że ten programista geniusz posiadł całą wiedzę programistyczną. Podobnie jak nauka języka obcego, nauka języka programowania wymaga dużo pracy i czasu. Dodatkowo, języki programowania czy techniki programowania zmieniają się znacznie częściej niż języki naturalne, dlatego też częścią programowania jest ciągłe uczenie się.

Ostatni mit natomiast mówi o tym, że dla każdego problemu programistycznego istnieje tylko jedno najlepsze rozwiązanie. Jeden problem można zazwyczaj rozwiązać na dziesiątki różnych sposobów. Wynika to z tego, że wiele aspektów programowania opiera się o personalne preferencje, np. wybór danego języka programowania, używanych bibliotek, czy stylu pisania kodu. W efekcie zazwyczaj nie możliwe jest jednoznaczne określenie, które rozwiązanie jest lepsze, szczególnie jeżeli wiele rozwiązań ma podobną wydajność. Istnieje jednak kilka reguł, z którymi zgadza się większość programistów. Pierwsza z nich mówi, że wolny działający kod jest lepszy niż szybki niedziałający kod[17]. Kolejna opiera się o zasadę DRY (nie powtarzaj się, ang. *Don't Repeat Yourself*), zalecającą unikanie różnego rodzaju powtórzeń wykonywanych przy programowaniu, np. używania tych samych fragmentów kodu w wielu miejscach. Ostatnia reguła mówi,

[15] Wiąże się to z popularnym na Zachodzie terminem data science[16], który łączy programowanie, analizę danych i wiedzę dziedzinową.

[17] Parafrazując Donalda Kuntha "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."

żeby tworzyć pisać programy w sposób modularny, czyli taki w którym każda funkcja spełnia tylko jedno i nie więcej zadanie, a złożone funkcje składają się z szeregu prostych funkcji.

1.2. Języki programowania

Głównym sposobem przekazywania instrukcji do komputera jest użycie języków programowania. Pozwalają one na precyzyjny zapis zadań, które następnie mają zostać wykonane przez komputer. Języki programowania składają się ze zbioru reguł syntaktycznych (składni) oraz semantyki. Składnia (forma) mówi o tym jakie symbole są dostępne w danym języku oraz jak te symbole mogą być łączone w większe struktury. Semantyka (treść) natomiast definiuje znaczenie poszczególnych symboli. W przeciwieństwie do języków naturalnych, języki programowania wymagają wysokiej precyzji. Mówiąc w języku naturalnym możemy popełnić jakiś błąd (np. gramatyczny czy składniowy) i nadal być łatwo zrozumianym przez otoczenie. Języki programowania nie akceptują takich błędów i nie są w stanie wykonać danego polecenia. Obecnie istnieją tysiące[18] języków programowania i każdego roku powstają nowe. Nie ma wśród nich jednego najlepszego, uniwersalnego języka programowania i w najbliższej przyszłości ten stan się nie zmieni. Jest to związane z bardzo szerokim zastosowaniem programowania w wielu dziedzinach czy problemach, które mają od siebie zupełnie różne wymagania. Przykładowe wymagania mogą dotyczyć np. szybkości wykonywanych obliczeń, łatwości pisania kodu, stabilności języka programowania, czy celu obliczeń. Do tego dochodzą również rożne kwestie historyczne i społeczne, jak na przykład preferowanie danego języka programowania przez osoby w danej branży. Obecnie wśród najpopularniejszych języków programowania można wymienić takie języki jak Java, C, Python, C++, Visual Basic .NET, JavaScript, C#, PHP, SQL, Objective-C, język asemblera, Perl, czy R. Języki programowania można podzielić na wiele różnych grup w zależności od przyjętych kryteriów. Poniżej wyjaśnionych jest kilka możliwych podziałów języków programowania.

Jednym z nich jest sposób wykonywania kodu - to czy kod w danym języku jest kompilowany czy też interpretowany. Kompilacja kodu (np. C czy Java) polega na jego tłumaczeniu do postaci języka maszynowego. W efekcie zapewnia to wysoką wydajność programu, ale za to kod jest ściśle powiązany z daną platformą sprzętową. Programowanie w językach kompilowanych jest zazwyczaj bardziej złożone i trudniejsze w nich jest odnajdywanie błędów (tzw. debugging). Interpretowane języki programowania, często również nazywane językami skryptowymi, (np. R czy Python) charakteryzuje to, że w momencie uruchomienia kod jest zamieniany na

[18] http://codelani.com/lists/languages.html

postać zrozumiałą dla komputera i od razu wykonywany. W efekcie można szybko zobaczyć efekt zmian. Wadą tego typu języków jest ich zmniejszona wydajność w porównany do języków kompilowanych.

Innym powszechnym podziałem języków programowania jest ich rozróżnianie na podstawie poziomu. Tutaj można wyróżnić języki od niskiego poziomu do wysokiego poziomu. Na najniższym poziomie jest język maszynowy, czyli taki w którym zapis programu wyrażony jest w postaci liczb binarnych. Powyżej są umieszczony jest język asemblera, w którym program jest zapisany poprzez serię instrukcji. Na najwyższym poziomie stawia się języki, które są wspomagane przez kompilator albo interpreter.

Języki programowania można też rozróżnić ze względu na paradygmat programowania. Definiuje on w jaki sposób w danym języku wykonywany jest przepływ sterowania czy też jak kod jest organizowany. Dwa podstawowe paradygmaty programowania to programowanie imperatywne i deklaratywne. Programowanie imperatywne (np. Fortran, C) opisuje proces wykonywania kodu jako sekwencję instrukcji zmieniających stan programu. Obejmuje ono inne paradygmaty, jak na przykład programowanie proceduralne czy obiektowe. Programowanie deklaratywne skupia się natomiast na warunkach jakie musi spełniać końcowe rozwiązanie, a nie na sekwencji kroków do jego stworzenia. W skład tej grupy wchodzi, między innymi, programowanie funkcyjne czy matematyczne. Niektóre języki mogą być zaklasyfikowane do kilku paradygmatów. Przykładowo R wspiera zarówno paradygmat funkcyjny, ale zawiera też możliwości programowania obiektowego.

1.3. R

W tej książce wprowadzenie do programowania opiera się o język R[19] (Rycina 1.2).

Rysunek 1.2.: Logo języka programowania R.

Wynika to z szeregu zalet tego języka:

[19] https://www.r-project.org/

- R jest bezpłatnym, otwartym oprogramowaniem, który można uruchomić na różnych systemach operacyjnych (Windows, Mac OS i Linux), zarówno na komputerach osobistych jak i na dużych klastrach obliczeniowych. W efekcie nie ma on finansowej bariery rozpoczęcia pracy, a kod napisany na jednym komputerze można również przenieść i uruchomić na innym sprzęcie.
- R jest językiem interpretowalnym, czyli wykonanie w nim komend nie wymaga kompilacji. Ten aspekt ułatwia szybsze zrozumienie działania tego języka.
- R posiada wiele wbudowanych narzędzi analizy i wizualizacji danych. Pozwala to na relatywnie szybkie osiąganie wymiernych efektów z korzystania z tego języka.
- R posiada tysiące dodatkowych rozszerzeń (zwanych pakietami) pozwalających na, między innymi, przetwarzanie różnorodnych danych, ich wizualizację, czy zaawansowane modelowanie. Oficjalnym portalem zawierającym dodatkowe pakiety R jest CRAN[20].
- R ma przyjazną społeczność użytkowników tego języka, zarówno online jak i spotykających się na żywo na tzw. meetupach.
- W celu ułatwienia pracy z R powstało również zintegrowane środowisko programistyczne RStudio, które wspomaga pisanie i analizę kodu w R.
- R został zaprojektowany jako narzędzie ułatwiające komunikację między różnymi językami programowania, głównie C oraz Fortran[21]. Obecnie R pozwala na łatwe łączenie kodu pochodzącego również z takich języków jak C++, Python, JavaScript, itd.
- R jest używany przez wiele małych firm, jak i wielkich korporacji, wliczając w to BBC, Facebook, Google, Microsoft, Mozilla, Netflix, T-Mobile, czy Uber[22].

Oczywiście, uniwersalny i idealny język nie istnieje:

- R jest językiem interpretowalnym, czyli wykonanie w nim komend nie wymaga kompilacji. W efekcie R nie jest najszybszym językiem programowania.
- Podobnie jak wiele innych języków, również R zawiera wiele niekonsekwencji, wynikających z wieloletniej ewolucji tego języka. W efekcie istnieje wiele specjalnych przypadków czy wyjątków, które warto znać (Burns, 2012).

Ta książka skupia się na prezentacji głównym konceptów programistycznych używając języka R. W sekcji 2.5 można znaleźć listę różnorodnych materiałów, książek, blogów, kursów, czy serwisów ułatwiających i wspomagających naukę R. Istnieje także wiele wprowadzających materiałów

[20] https://cran.r-project.org/
[21] https://www.youtube.com/watch?v=_hcpuRB5nGs
[22] https://github.com/ThinkR-open/companies-using-r

do nauki innych języków. Przykładowo, osoby zainteresowane nauką Pythona mogą skorzystać z istniejących książek (Gries et al. (2017) oraz Guzdial and Ericson (2016)), czy też kursów Software Carpentry[23] oraz Python Course[24]. W pracy programistycznej przydaje się również często znajomość linii komend. Tutaj również można użyć materiałów z kursu Software Carpentry[25] lub książki The Unix Workbench[26] (Kross, 2017).

1.4. Zadania

1) Pomyśl do czego jesteś w stanie wykorzystać programowanie w swoim życiu zawodowym lub prywatnym?
2) Zastanów się nad mitami związanymi z programowaniem. Czy jesteś w stanie wskazać jakieś mity nie wymienione powyżej?
3) Wybierz trzy języki programowania z listy wymienionej w tym rozdziale i poszukaj informacji o nich. Do czego są one stosowane? Jakie mają wady i zalety?

[23] http://swcarpentry.github.io/python-novice-inflammation/
[24] https://www.python-course.eu/python3_course.php
[25] https://swcarpentry.github.io/shell-novice/
[26] https://seankross.com/the-unix-workbench/

Część I.

Podstawy

2. Start R

Wykonywanie kodu w języku interpretowalnym, jakim jest R, może odbywać się poprzez wpisanie polecenia w oknie konsoli (zwanej też terminalem) i jego uruchomienie[1]. Komendy są najpierw sprawdzanie pod kontekstem ich poprawności. Polega to na określeniu, np. czy podana funkcja lub inny obiekt istnieje, czy nie zostały użyte niedozwolone znaki, lub czy wszystkie nazwiasy czy cudzysłowia zostały zamknięte. Języki programowania są w tym aspekcie bardziej bezwzględne niż języki naturalne - nie potrafią one zrozumieć wyrażeń zawierających nawet niewielkie błędy takie jak, np. użycie dużej litery zamiast małej.

2.1. Wyrażenia

Podstawowe działania arytmetyczne, dodawanie, odejmowane, mnożenie i dzielenie, są również często używane w wielu językach programowania. Dla każdej z tych operacji istnieje odpowiedni operator w R. Operatorem dodawania jest `+`.

```
2 + 2
#> [1] 4
```

Operatorem odejmowania jest `-`.

```
1 - 3
#> [1] -2
```

Operatorem mnożenia jest `*`.

```
5 * 5
#> [1] 25
```

[1] To jest tzw. tryb interaktywny. Isnieje również tryb skryptowy, o którym więcej informacji można znaleźć w kolejnym rozdziale.

Operatorem mnożenia jest `/`.

```
42 / 5
#> [1] 8.4
```

Wszystkie powyższe operacje można wykonać poprzez ich wpisanie w oknie konsoli R i naciśnięcie klawisza Enter.

2.2. Obiekty

> "Dwa slogany są pomocne w zrozumieniu obliczeń w R: 1. Wszystko co istnieje jest obiektem. 2. Wszystko co się dzieje jest wywołaniem funkcji." John Chambers

Powyższy cytat sugeruje dwa najważniejsze elementy języka R: obiekty i funkcje. Zrozumienie w jaki sposób się je tworzy i zmienia będzie w związku z tym, konieczną wiedzą osób piszących w tym języku.

2.2.1. Operator przypisania

Nadanie wartości do obiektu wykonuje się używając operatora przypisania[2]. R posiada trzy operatory przypisania, które mają niemal identyczne działanie[3]: `=`, `<-`, `->`. Warto wybrać jeden z tych operatorów i konsekwentnie używać go pisząc kod. W tej książce jako główny operator przypisania będzie używany znak `=`.

W poniższej linii stworzony jest nowy obiekt, o nazwie `x`, który zawiera wartość `7`.

```
x = 7
```

Można to sprawdzić wpisując nazwę tego obiektu.

```
x
#> [1] 7
```

[2] Jest to pewne uproszczenie - https://adv-r.hadley.nz/names-values.html#binding-basics.

[3] Więcej informacji na temat różnic w działaniu tych operatorów można znaleźć na stronie https://stackoverflow.com/questions/1741820/what-are-the-differences-between-and-in-r.

Operatory przypisania może również posłużyć do nadania wartości z jednego obiektu do drugiego. Poniżej nowy obiekt `y` przyjmuje wartość od obiektu `x`.

```
y = x
y
#> [1] 7
```

Język R przechowuje i przetwarza wszystkie obiekty w pamięci komputera (RAM). Wpływa to na zwiększoną wydajność i elastyczność obliczeń, ale jednocześnie powoduje to ograniczenie wielkości obiektów na jakich można pracować. Istnieje równocześnie szereg strategii[4] jak postępować z większymi zbiorami danych, które nie mieszczą się w RAMie (Peng et al., 2017).

2.2.2. Działania na obiektach

Każdy stworzony obiekt w R może być następnie używany do kolejnych operacji, a w efekcie też tworzenia nowych obiektów. W poniższych czterech przypadkach obiekt `x` został przetworzony używając operatorów dodawania, odejmowania, mnożenia oraz dzielenia, a nowe obiekty powstały jako wyniki tych obliczeń.

```
z1 = x + 3
z1
#> [1] 10
z2 = x - 5
z2
#> [1] 2
z3 = x * 2
z3
#> [1] 14
z4 = x / 4
z4
#> [1] 1.75
```

Część języków programowania, np. C, wymaga zadeklarowania zmiennej przed jej użyciem poprzez podanie jej nazwy i typu. Wybór typu zmiennej w tych językach może mieć widoczne konsekwencje.

[4] https://bookdown.org/rdpeng/RProgDA/working-with-large-datasets.html

> Przykładowo, jeżeli obiekt `x` zostanie zadeklarowany jako liczba całkowita (integer), wynikiem dzielenia `x / 4` będzie `1` zamiast `1.75`.

Działania na obiektach mogą też się odbywać używając innych operatorów oraz różnorodnych funkcji. Przykładowo, operator zapisywany jako `%%` to modulo, którego celem jest określanie reszty z dzielenia.

```
z5 = x %% 3
z5
#> [1] 1
```

Operator `%/%` przestawia dzielenie całkowite.

```
z6 = x %/% 3
z6
#> [1] 2
```

Operator `^` natomiast wykonuje podniesienie wartości obiektu do wybranej potęgi.

```
z7 = x^2
z7
#> [1] 49
```

Odwrotnością potęgowania jest pierwiastkowanie. W R nie istnieje do tego celu specjalny operator, ale zawiera on specjalną funkcję `sqrt()`.

```
z8 = sqrt(x)
z8
#> [1] 2.65
```

Często używaną funkcją w R jest też `c()`. Ta funkcja łączy krótsze wektory w dłuższe wektory.

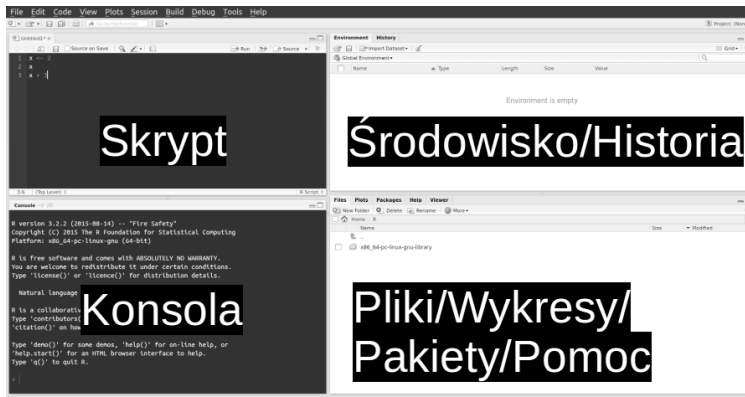
```
z9 = c(z2, z4, z8)
z9
#> [1] 2.00 1.75 2.65
```

Operatory użyte w tym rozdziale, np. `+`, `*`, `^`, `%%` to też są funkcje, ale zapisane w skrótowej formie ułatwiającej z nimi pracę. Te operatory można też użyć jako normalne funkcje poprzez dodanie znaku zwanego grawisem - "`, `np.2 + 2`można też zapisać jako`‘+‘(2, 2)‘.

2.3. IDE

RStudio to zintegrowane środowisko programistyczne (ang. *Integrated Development Environment*, IDE) dla R. Zawiera ono bardzo wiele użytecznych funkcjonalności, tj. wbudowany edytor, podświetlanie składni, automatyczne uzupełnianie kodu i wiele innych.

RStudio to nie jest to samo co R. R jest językiem programowania, podczas gdy RStudio to aplikacja ułatwiająca pisanie kodu. Możliwe jest używanie R bez RStudio, ale RStudio bez R nie pełni już swojej roli. Częstą analogią jest porównanie samochodowe, w którym R jest opisywany jako silnik a RStudio jako deska rozdzielcza.



Rysunek 2.1.: Okno RStudio z opisaną funkcjonalnością każdej z jego części.

Typowa praca w RStudio często polega na wpisywaniu poleceń do pliku tekstowego widocznego w części skryptowej (Rycina 2.1), a następnie wykonywaniu kolejnych linii kodu w oknie konsoli używając skrótu klawiaturowego CTRL+ENTER (więcej przydatnych skrótów klawiaturowych można znaleźć w tabeli 2.1). Efektem wykonywania funkcji może być powstanie nowych obiektów, które można zobaczyć w oknie “środowiska” lub też wyświetlenie grafik, które można zobaczyć w oknie “wykresu”.

Tabela 2.1.: Podstawowe skróty klawiaturowe w RStudio

Skrót	Wyjaśnienie
Ctrl+Enter	wykonuje wybraną linię kodu w skrypcie R
Tab	uzupełnia kod (podaje pasujące możliwości)
F1	wyświetla plik pomocy dla wybranej funkcji
Ctrl+Shift+C	ustawia wybrane linie jako komentarz/odkomentuj fragment kodu
strzałka Góra/Dół (w oknie konsoli)	wybiera wcześniej wpisany kod
Esc	przerywa niedokończoną operację
Shift+Alt+K	wyświetla listę skrótów klawiaturowych

Dobrą praktyką pracy z R w RStudio jest też używanie projektów RStudio (ang. *RStudio projects*). Projekt jest to folder zawierający wszystkie skrypty i pozostałe pliki powiązane z jakimś zadaniem (np. analizą danych, czy stworzeniem nowego pakietu R). Ułatwia on przenoszenie kodu pomiędzy różnymi komputerami, a także daje dostęp do szeregu dodatkowych możliwości w RStudio.

Aby stworzyć pierwszy projekt RStudio, należy:

1. Kliknąć `File -> New Project`.
2. Wybrać `New Directory`.
3. Wybrać `New Project`.
4. Podać nazwę nowego projektu, np. "programowanie1" oraz wybrać miejsce na dysku, gdzie ma się nowy projekt znajdować.
5. Jeżeli możliwe, to wybrać też opcję `Create a git repository`.
6. Kliknąć `Create Project`.

```
#> Registered S3 method overwritten by 'rvest':
#>   method            from
#>   read_xml.response xml2
```

2.4. Styl

Poniżej znajdują się podstawowe porady dotyczące stylu pisania kodu. Więcej wskazówek można znaleźć na w poradniku stylu RStudio[5] oraz

[5]https://style.tidyverse.org/

poradniku stylu Google[6]. Oba te poradniki nie są identyczne i czasami zawierają sprzeczne porady. Najważniejsze jest, aby wybrać jeden odpowiadający piszącemu kod styl i się go konsekwentnie trzymać.

2.4.1. Nazwy obiektów

Istnieje wiele konwencji nazywania obiektów[7]. Najczęściej używaną konwencją w R jest tzw. "snake case"[8]. Polega ona na tworzeniu nazw obiektów składających się ze słów połączonych znakiem podkreślenia (_). Ważne, żeby nazwy obiektów ułatwiały zrozumienie ich zawartości.

```
# obiekt
bok_a
bok_b

# funkcja
pole_prostokata
```

Nazwa obiektu nie może zaczynać się od liczby, ani nie może używać specjalnych symboli, tj. `^`, `!`, `$`, `@`, `+`, `-`, `/`, czy `*`. Dodatkowo należy uważać, żeby nowa nazwa obiektu nie nadpisała istniejącego obiektu lub funkcji. Nie powinno nazywać się obiektów tak jak istniejące funkcje, np. `c`, `t`, `table`, itd.

2.4.2. Odstępy

Odstępy pełnią bardzo ważną funkcję przy pisaniu kodu, podobnie jak odstępy przy pisaniu tekstu. Wyobraź sobie czytanie powieści, w której nie ma żadnych odstępów między słowami czy rozdziałami. Często mówi się, że "kod musi oddychać" - odstępy zwiększają czytelność kodu i pozwalają na jego szybsze zrozumienie oraz ułatwiają naprawienie występujących błędów.

Odstępy można uzyskać poprzez użycie spacji. Spacje powinny być użyte po przecinkach, ale nigdy przed nimi. Dodatkowo, większość operatorów (np. `=`, `+`, `-`, `==`) powinna być otoczona przez spacje.

[6] https://google.github.io/styleguide/Rguide.xml
[7] https://en.wikipedia.org/wiki/Naming_convention_(programming)
[8] https://en.wikipedia.org/wiki/Snake_case

```
# Zalecane
srednia = mean(wartosc, na.rm = TRUE)
pole = bok_a * bok_b

# Niewskazane
srednia=mean ( wartosc,na.rm=TRUE )
pole=bok_a*bok_b
```

Spacje należy również używać do tworzenia wcięć - każde z nich powinno się składać z dwóch spacji.

```
# Zalecane
moja_funkcja = function(x, y, z){
  pod = y / z
  wynik = x * pod
  wynik
}

# Niewskazane
moja_funkcja = function(x, y, z){
pod = y / z
wynik = x * pod
wynik
}
```

Warto także ograniczać długość każdej linii kodu, żeby nie przekraczała ona ok. 80 znaków. Dzięki temu możliwe jest szybkie przeczytanie kodu czy też jego wydrukowanie.

```
# Zalecane
bardzo_wazny_wynik = moja_bardzo_wazna_funkcja("pierwszy argument",
                                               b = "drugi argument",
                                               c = "trzeci argument")

# Niewskazane
bardzo_wazny_wynik = moja_bardzo_wazna_funkcja("pierwszy argument", "drugi argument", "trzeci a
```

2.4.3. Komentarze

Komentarze służą do wyjaśniania istotnych elementów kodu. Do komentowania w języku R służy operator `#`.

```
# Mój komentarz
```

2.4.4. Nazwy plików

Nazwy plików powinny spełniać trzy wymagania - być łatwe (i) do odczytania przez komputer, (ii) do odczytania przez człowieka, (iii) do posortowania.

Nazwy plików nie powinny zawierać spacji, znaków specjalnych (np. !, %, *), znaków diakrytycznych (np. ć, Ł, ź). Warto też aby nazwy plików składały się tylko z małych liter.

```
# Zalecane
obliczanie-sredniej.R
pomiary-temperatury.csv

# Niewskazane
Obliczanie Średniej.R
pomiaryTemperatury!.csv
```

Podobnie jak nazwy obiektów, również nazwy plików powinny opisywać ich zawartość.

```
# Zalecane
obliczanie-sredniej.R
pomiary-temperatury.csv

# Niewskazane
kod.R
dane.csv
```

Dodatkowo wskazane jest dodanie wartości numerycznych przed nazwą pliku, jeżeli pliki mają jakąś kolejność.

```
# Zalecane
01_przygotowanie-danych.R
02_obliczanie-sredniej.R

# Niewskazane
przygotowanie-danych.R
obliczanie-sredniej.R
```

Kodowanie znaków (ang. *character encodings*) jest to sposób sposób prezentacji znaków. Istnieje szereg różnych standardów kodowania znaków. Standard ASCII przyporządkowuje liczbom z zakresu 0-127 litery alfabetu angielskiego, cyfry, znaki przestankowe i inne symbole oraz polecenia. Firma Microsoft stworzyła dodatkowo cały szereg standardów dla różnych języków. Przykładowo do obsługi języków środkowoeuropejskich istnieje wersja oznaczona jako Windows-1250 (lub CP1250). Alternatywnie do systemu Microsoftu powstał też zbiór standardów ISO, przykładowo ISO-8859-2 dla języków środkowoeuropejskich. W efekcie oznacza to, że otworzenie tekstu z innego komputera, na komputerze z "polskim" kodowaniem znaków może spowodować pojawienie się tzw. "krzaczków". Aby uniknąć takiej sytuacji powstał system kodowania UTF-8, który zawiera w sobie ponad milion różnych znaków. Jest on obecnie zalecanym standardem na całym świecie.

2.4.5. Daty

Istnieje wiele sposobów zapisu dat[9], co może powodować różnorodne problemy przy programowaniu oraz analizie danych. Z ratunkiem w tej kwestii przychodzi norma ISO 8601[10], która definiuje daty kalendarzowe jako *YYYY-MM-DD*, czyli *ROK-MIESIĄC-DZIEŃ*.

```
# Zalecane
2019-06-02

# Niewskazane
wszelkie inne
```

2.5. Dodatkowe materiały

Polskie książki:

- http://www.biecek.pl/R/ (Biecek, 2014)
- http://www.gagolewski.com/publications/programowanier/ (Gagolewski, 2016)
- https://helion.pl/ksiazki/jezyk-r-kompletny-zestaw-narzedzi-dla-analitykow-danych-hadley-wickham-garrett-grolemund,jezrko.htm#format/d (Wickham and Grolemund, 2016)

[9] https://xkcd.com/1179/
[10] https://en.wikipedia.org/wiki/ISO_8601

- https://helion.pl/ksiazki/wydajne-programowanie-w-r-praktyczny-przewodnik-po-lepszym-programowaniu-gillespie-colin-lovelace-robin,a_0491.htm#format/d (Gillespie and Lovelace, 2016)
- https://bookdown.org/nowosad/Geostatystyka/ (Nowosad, 2019)
- http://www.enwo.pl/przetwarzanie/index.html (Czernecki, 2018)

Angielskie książki:

- https://rstudio-education.github.io/hopr/ (Grolemund, 2014)
- https://r4ds.had.co.nz/ (Wickham and Grolemund, 2016)
- https://csgillespie.github.io/efficientR/ (Gillespie and Lovelace, 2016)
- https://adv-r.hadley.nz (Wickham, 2014)
- https://geocompr.robinlovelace.net/ (Lovelace et al., 2019)

Blogi:

- Agregator blogów dotyczących R - https://www.r-bloggers.com/
- Polski blog opisujący kwestie analizy danych w R, wizualizacji, oraz edukacji - http://smarterpoland.pl/
- Polski blog pokazujący zastosowanie R do analizy i wizualizacji danych - http://szychtawdanych.pl/

Kursy:

- Polskie tłumaczenie pakietu R służącego do nauki tego języka - https://github.com/dabrze/swirl
- Lista kursów dotyczących R na platformie Coursera - https://www.coursera.org/courses?query=r
- Lista kursów dotyczących R na platformie edX - https://www.edx.org/course?search_query=r

Pisanie kodu oraz jego dokumentowanie opiera się w znacznym stopniu na wprowadzaniu znaków na klawiaturze do komputera. Warto jest więc aby robić to w sposób efektywny[11], czyli taki w którym używamy wszystkich palców u rąk a nasz wzrok nie jest skupiony na klawiaturze. Takie pisanie nazwa się pisaniem bezwzrokowym (ang. *touch typing*). Pisanie bezwzrokowe ma szereg reguł, które wymagają przestawienia się ze starych nawyków oraz pewnego treningu. Na szczęście istnieje wiele internetowych zasobów, które ułatwiają naukę takiego pisania, między innymi strona TypingClub[12].

Serwisy internetowe:

- Wyszukiwarki internetowe są nieocenionym narzędziem wspierającym programowanie - https://rseek.org/, https://duckduckgo.com/, https://www.google.com/, https://www.bing.com/, itd.

- Serwis społecznościowy zawierający pytania i odpowiedzi dotyczące różnych języków programowania w tym R - https://stackoverflow.com. Pytania dotyczące R można znaleźć pod adresem https://stackoverflow.com/questions/tagged/r. Przed zadaniem nowego pytania warto wyszukać czy nie zostało ono zadane wcześniej a następnie przeczytać wątek dotyczący tworzenia nowych pytań - https://stackoverflow.com/questions/5963269/how-to-make-a-great-r-reproducible-example
- Twitter jest miejscem, w którym można znaleźć zarówno nowości z języka R, jak również odpowiedzi na pytania dotyczące tego języka - https://twitter.com/. Kwestie związane z R są opatrzone hasztagiem `#rstats`, natomiast kwestie przestrzenne w R są opisywane hasztagami `#rspatial` oraz `#geocompr`
- Elektroniczny biuletyn R Weekly zbierający co tydzień nowości związane z r - https://rweekly.org/
- Lista emailowa dotycząca R - https://stat.ethz.ch/mailman/listinfo/r-help
- Lista emailowa dotycząca kwestii przestrzennych w R - https://stat.ethz.ch/mailman/listinfo/r-sig-geo
- Forum dotyczące kwestii R i RStudio - https://community.rstudio.com/

Meetups (spotkania początkujących i zaawansowanych użytkowników R):

- Poznań - https://www.meetup.com/pl-PL/Poznan-R-User-Group-PAZUR/
- Warszawa - https://www.meetup.com/pl-PL/Spotkania-Entuzjastow-R-Warsaw-R-Users-Group-Meetup/
- Wrocław - https://www.meetup.com/Wroclaw-R-Users-Group/
- Kraków - https://www.meetup.com/erkakrakow/
- Trójmiasto - https://www.meetup.com/Trojmiejska-Grupa-Entuzjastow-R/

2.6. Zadania

Rozwiązując poniższe zadania oraz pozostałe zadania z tej książki staraj się stosować do stylu podanego w sekcji 2.4.

1) Przejrzyj poniższą listę poleceń. Spróbuj określić uzyskane wyniki bez wykonywania kodu w R.

```
x = 7
y = -2
x + 3
```

```
y - 5
x * 2
y / 4
x %% 3
x %/% 3
y ^ 2
y ^ x
```

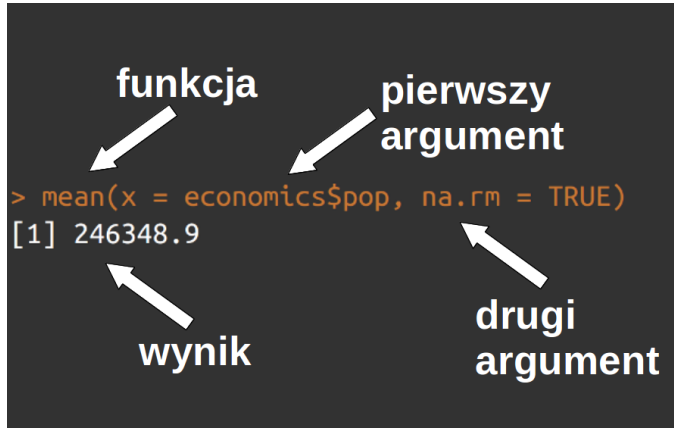
2) Jedziesz na krótkie wakacje i planujesz na nie zabrać 500 EUR. Aktualny kurs kupna EUR wynosi 4,31. Ile PLN musisz wydać? Wylicz to w R.
3) Masz trapez o długości podstaw `a = 5` i `b = 6` oraz wysokości `h = 3`. Stwórz nowy obiekt `pole_trapezu`, który zawiera obliczone pole tego trapezu.
4) Wraz z grupą znajomych planujesz zamówić pizzę z dostawą i macie na to przeznaczonych 50 PLN. Pizza o średnicy 30 cm kosztuje 23,5 PLN, a pizza o średniczy 50 cm kosztuje 50 PLN. Wylicz w R, czy bardziej opłaca się kupno dwóch małych pizz czy jednej dużej.

3. Funkcje

Funkcje to programy, który przyjmują pewne argumenty, przetwarzają je i zwracają jakiś wynik. Są one zbudowane z dostępnych elementów języka programowania jak i też z innych dostępnych funkcji. Funkcje mogą służyć do wielu celów, od prostych odliczeń arytmetycznych, poprzez przetwarzanie tekstu, tworzenie wykresów i map, aż do bardziej złożonych i specjalistycznych procedur. Ich celem jest ułatwienie pracy programistycznej i zwiększenie czytelności kodu. Zamiast wielokrotnie powtarzać te same linie kodu, możliwe jest napisanie funkcji raz, a następnie użycie jej wiele razy.

3.1. Struktura funkcji

Funkcje są reprezentowane w R jako specjalne obiekty, które można uruchomić poprzez dodanie do ich nazwy nawiasów okrągłych. Przykładowo, funkcja `mean()` wylicza średnią. Może ona przyjąć kilka różnych argumentów, czyli pewnych obiektów lub parametrów wejściowych. W poniższym przykładzie do funkcji `mean()` zostały podane dwa argumenty (rycina 3.1). Pierwszy argument nazywa się `x` i przyjmuje on wektor numeryczny `economics$pop`, drugi argument nazywa się `na.rm` i został on ustalony na `TRUE`.



Rysunek 3.1.: Przykład struktury funkcji w R.

W efekcie działania funkcji otrzymano wynik - `246348.9` - który jest średnią wartością w zadanym wektorze.

3.2. Wbudowane funkcje

R posiada wiele wbudowanych funkcji, które znacznie ułatwiają wykonywanie bardziej złożonych operacji. Pierwsze z funkcji, w tym `c()`, zostały już poznane w sekcji 2.2.2. Ta funkcja pozwala na łączenie kolejnych obiektów rozdzielonych przecinkami. Poniższy obiekt, `x` jest w efekcie wektorem zawierającym trzy wartości `8.2`, `10.3` oraz `12.0`.

```
x = c(8.2, 10.3, 12.0)
x
#> [1]  8.2 10.3 12.0
```

Wyliczenie średniej z tych trzech wartości wymaga ich zsumowania, a następnie podzielenia uzyskanej wartości przez liczbę wartości.

```
(8.2 + 10.3 + 12.0) / 3
#> [1] 10.2
```

W przypadku jednak, gdy chcemy dodać czwartą, piątą, itd. wartość należy zmieniać kod w co najmniej dwóch miejscach. Konieczne jest dodanie nowej wartości do zsumowania, a następnie zmianę wartości określającej liczbę elementów.

Zamiast tych dwóch operacji, można wykonać tylko jedną zmianę w obiekcie `x`, a następnie przetworzyć go używając funkcji wbudowanych w R - `sum()` oraz `length()`. Pierwsza z nich sumuje wartości z wektora, druga natomiast zwraca liczbę elementów w wektorze.

```
sum(x) / length(x)
#> [1] 10.2
```

Powyższy kod można też dalej uprościć, poprzez użycie wbudowanej w R funkcji do liczenia średniej - `mean()`.

```
mean(x)
#> [1] 10.2
```

Jej użycie powoduje, że wystarczy wykonać tylko jedną zmianę w obiekcie `x`, aby uzyskać poprawny wynik, a dodatkowo napisanie tego obliczenia wymaga napisania znacznie krótszego kodu.

Funkcje można używać w celu wyświetlenia oczekiwanego rezultatu, ale także, aby na podstawie wyniku funkcji tworzyć nowe obiekty, takie jak `y` poniżej.

```
y = mean(x)
y
#> [1] 10.2
```

3.3. Kolejność wykonywania funkcji

Wykonywanie funkcji w R odbywa się linia po linii, od góry do dołu.

```
a = 4
b = 5
a2 = a^2
b2 = b^2
```

R pozwala na dwa podstawowe sposoby łączenia działania wielu funkcji[1]. Pierwszy z nich polega na tworzeniu pośrednich obiektow jako wyników działania pojedynczych funkcji.

```
suma_a2b2 = sum(a2, b2)
przekatna = sqrt(suma_a2b2)
przekatna
#> [1] 6.4
```

Drugi sposób opiera się o zagnieżdzanie funkcji. W tej sytuacji najpierw wykonywana jest funkcja w środku, na następnie kolejne funkcje coraz bliżej brzegu.

```
przekatna = sqrt(sum(a2, b2))
przekatna
#> [1] 6.4
```

[1] Istnieje też szereg dodatkowych sposobów, wśród których najpopularniejszy polega na używaniu operatora `%>%` z pakietu **magrittr** (Bache and Wickham, 2014).

3.4. Dokumentacja funkcji

Każda wbudowana funkcja w R posiada swoją dokumentację[2]. Można ją wyświetlić poprzez dodanie znaku zapytania przed nazwą funkcji, a następnie wykonanie tej linii kodu.

```
?mean
```

Alternatywnie, w RStudio możliwe jest użycie skrótu F1 gdy kursor znajduje się na nazwie funkcji.

Dokumentacja każdej funkcji, zwana inaczej plikiem pomocy, ma zazwyczaj podobną strukturę.

- W lewym górnym rogu znajduje się nazwa funkcji (`mean`) oraz nazwa pakietu z którego dana funkcja pochodzi (`base`).
- Poniżej znajduje się tytuł funkcji oraz jej krótki opis.
- Kolejnym elementem jest budowa funkcji (*Usage*), która skrótowo opisuje z jakich argumentów składa się dana funkcja. Np. funkcja `mean()` przyjmuje argument `x`, `trim`, oraz `na.rm`. Dla argumentów `trim` oraz `na.rm` są także ustalone ich domyślne wartości. Dodatkowo, widoczny jest argument w postaci wielokropka (...).
- Argumenty funkcji są również wypisane oraz skrótowo wyjaśnione. Przykładowo, `x` musi być obiektem R o typie numerycznym (który łączy typ liczb całkowitych i zmiennoprzecinkowych), logicznym, `date`, `date-time`, lub `time interval`.
- Część *Value* (lub *Details*) opisuje szczegóły wykonywanej funkcji.
- Inne możliwe elementy to np. *References* odnoszący się do artykułu czy książki opisującej daną funkcję lub metodę, czy też *See also* zawierający odnośniki do innych, podobnych funkcji.
- Jeden z najważniejszych elementów pliku pomocy znajduje się na samym końcu - są to przykłady (*Examples*). Jeżeli nie jesteśmy pewni jak dana funkcja działa warto zacząć od skopiowania przykładów a następnie ich wykonania.

Czytanie dokumentacji wymaga pewnej wprawy i doświadczenia. Nie bój się używać innych źródeł pomocy (zobacz sekcję 2.5), jeżli potrzebujesz zrozumieć działanie danej funkcji.

3.5. Pakiety

Pakiet to zorganizowany zbiór funkcji, który rozszerza możliwości R. Pakiety oprócz kodu zawierają szereg dodatkowych istotnych elementów, ta-

[2]Niektóre zbiory danych również posiadają swoje pliki pomocy.

kich 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 jedno centralne repozytorium (CRAN, ang. *the Comprehensive R Archive Network*), które zawiera oficjalne wersje pakietów R. Wersje deweloperskie (rozwojowe) często można znaleźć na platformie GitHub[3].

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.

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[4] dla systemu Windows czy też narzędzia Xcode dla Mac OS. 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.

[3] https://github.com/

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

```
library(stringr)
```

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"
```

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.

3.6. Algorytmy

Algorytm to zbiór kroków powadzących do uzyskania określonego celu. Algorytmy można porównać do przepisu kucharskiego, w którym opisany jest szereg czynności aby uzyskać konkretną potrawę. Podobnie jak w przepisie kucharskim, algorytmy wymagają posiadania odpowiednich składników - danych wejściowych w o pewnej strukturze.

Tworzenie nowych algorytmów często zaczyna się od narysowania schematu procedury działania lub też pseudokodu. Kolejnym krokiem jest zapisanie tego algorytmu w wybranym języku lub językach programowania w formie skryptu (sekcja 3.7) lub funkcji (sekcja 3.8).

3.7. Tworzenie skryptów

Skrypt w R to plik testowy z rozszerzeniem `.R`, który zawiera szereg linii kodu w celu uzyskania konkretnego efektu. Może on zawierać zaledwie kilka jak i setki linii kodu w zależności od złożoności postawionego problemu. Zobaczmy jak wyglądają skrypty na prostym przykładzie - przeliczania wartości ze skali Fahrenheita na skalę Celsjusza. Otrzymaliśmy informację, że w "mieście A" temperatura w stopniach Fahrenheita wynosi 75.

```
miasto_a = 75
```

Pierwszym naszym krokiem powinno być dowiedzenie się jaka jest relacja pomiędzy skalą Fahrenheita na skalą Celsjusza.

$$T_{Celsjusz} = \frac{T_{Fahrenheit} - 32}{1.8}$$

Następnie powyższy wzór można przepisać do postaci kodu w języku R oraz podstawić do niego wartość temperatury w stopniach Fahrenheita w mieście A. Ostatnim etapem jest wyświetlenie uzyskanego wyniku - temperatura w mieście A wynosi ok. 24 stopnie Celsjusza.

```
miasto_a_c = (miasto_a - 32) / 1.8
miasto_a_c
#> [1] 23.9
```

Powyższe kroki można również zapisać do pliku tekstowego.

```
# plik przeliczanie-temp.R
miasto_a = 75
miasto_a_c = (miasto_a - 32) / 1.8
miasto_a_c
```

Co można zrobić jeżeli mamy więcej podobnych pomiarów, które chcemy wykonać? Najprostszą opcją jest użycie kopiuj/wklej i powielenie tego samego kodu, a później naniesienie małych zmian, np. nazw obiektów.

```
miasto_a = 75
miasto_b = 110
miasto_c = 0
miasto_a_c = (miasto_a - 32) / 1.8
miasto_b_c = (miasto_b - 32) / 1.8
miasto_c_c = (miasto_c - 32) / 1.8
```

Powyższe podejście jest poprawne, ale ma ono kilka wad:

- Łatwo jest o popełnie jakiegoś prostego błędu lub literówki podczas adaptacji kodu (np. można zapomnieć zmienić nazwę jakiejś zmiennej).
- Jeżeli obliczenia zajmują więcej niż kilka linii kodu - wówczas kopiowanie go znacznie powiększa tworzony skrypt i utrudnia jego czytelność.
- Poprawienie kodu w przypadku zauważenia błędu w procedurze obliczeniowej jest czasochłonne.

To podejście jest też niezgodne z jedną z najważniejszych reguł w programowaniu - regułą DRY (Nie powtarzaj się, ang. *Don't Repeat Yourself*). Zamiast tworzenia skryptu w oparciu o kopiuj/wklej lepiej pomyśleć nad zbudowaniem odpowiedniej funkcji[5].

3.8. Budowanie funkcji

Funkcje pozwalają na automatyzację często używanych obliczeń. Formalnie funkcje składają się z trzech elementów: listy argumentów (ang. *formals*), ciała funkcji (ang. *body*) oraz środowiska (ang. *environment*). Pierwsze dwa elementy ustala twórca funkcji, natomiast środowisko jest określane na podstawie tego, gdzie dana funkcja została zdefiniowana. Dodatkowo każda funkcja ma swoją nazwę.

```
moja_funkcja = function(x, y, z){
  pod = y / z
  wynik = x * pod
  wynik
}
```

Lista argumentów wymienia obiekty wejściowe funkcji.

[5] Wickham and Grolemund (2016) radzą tworzyć nowe funkcje, gdy ten sam kod powtarza się co najmniej trzy razy.

```
formals(moja_funkcja)
#> $x
#>
#>
#> $y
#>
#>
#> $z
```

Ciało zawiera kod danej funkcji.

```
body(moja_funkcja)
#> {
#>     pod = y/z
#>     wynik = x * pod
#>     wynik
#> }
```

Środowisko określa, gdzie dana funkcja jest zlokalizowana.

```
environment(moja_funkcja)
#> <environment: R_GlobalEnv>
```

Przykładowa funkcja odpowiadająca problemowi z poprzedniej sekcji może wyglądać w poniższy sposób:

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

Nowa funkcja nazywa się `konwersja_temp()` oraz posiada tylko jeden argument `temperatura_f`. Ciało funkcji zawiera natomiast wzór potrzebny do obliczeń przepisany do R. Ważne jest to, że obiekt użyty wewnątrz funkcji (`temperatura_f`) jest taki sam jak wejściowy argument.

Po stworzeniu funkcji warto sprawdzić czy jej działanie odpowiada naszym oczekiwaniom.

```
konwersja_temp(75)
#> [1] 23.9
konwersja_temp(110)
#> [1] 43.3
konwersja_temp(0)
#> [1] -17.8
konwersja_temp(c(0, 75, 110))
#> [1] -17.8  23.9  43.3
```

3.9. Komunikaty

Oprócz wyniku danej operacji R może wyświetlić kilka rodzajów komunikatów. Trzy podstawowe z nich to:

1. Błędy (ang. *errors*)
2. Ostrzeżenia (ang. *warnings*)
3. Wiadomości (ang. *messages*)

Błędy oznaczają, że wykonanie danej funkcji nie może być kontynuowane i przerwane jest jej działanie. Przykładowo, w poniższym kodzie podjęta została próba wyliczenia logarytmu naturalnego z tekstu `"abecadło"`. Takie obliczenie nie jest możliwe, w efekcie pojawił się komunikat błędu a kod nie został wykonany.

```
log("abecadło")
#> Error in log("abecadło"): non-numeric argument to mathematical function
```

Ostrzeżenia zazwyczaj występują kiedy nastąpił jakiś problem z wykonaniem funkcji, ale jej działanie mogło być dokończone. Często sugerują one użytkownikowi, aby dokładnie przyjrzał się wykonywanej funkcji i upewnił się czy na pewno ustala on odpowiednie wartości dla argumentów funkcji. Poniżej została podjęta próba wyliczenia logarytmu naturalnego dla wartości ujemnej. W efekcie pojawił się komunikat błędu, który mówi, że w wyniku zostały stworzone wartości `NaN` (ang. *Not a Number*).

```
log(-1)
#> Warning in log(-1): NaNs produced
#> [1] NaN
```

Wiadomości pojawiają się, aby przekazać użytkownikowi jakąś informację.

```
inna_funkcja(15)
#> Chcę ciebie o czymś poinformować.
#> [1] 2.71
```

Opis tworzenia komunikatów błędu, ostrzeżenia i wiadomości można znaleźć w rozdziale 10.

3.10. Zadania

1) Zobacz jak wygląda plik pomocy funkcji `mean()`. Wykonaj zawarte w nim przykłady. Co przedstawiają uzyskane wyniki?
2) Zainstaluj pakiet **magrittr**. Spróbuj użyć operatora `%>%` z tego pakietu na przykładzie z sekcji 3.3 dotyczącym wyliczania przekątnej prostokąta.
3) Stwórz nowy plik skryptu R nazywający się `01_zadania-funkcje.R`. W tym pliku, stwórz nowy obiekt `poznan`, który przyjmuje wartość `8.4`, napisz przeliczenie wartości tego obiektu ze stopnii Celsjusza na stopnie Fahrenheita, a następnie wyświetl uzyskany wynik. Uwaga: pamiętaj o ustawieniu odpowiedniego kodowania znaków dla tego nowego pliku.
4) Stwórz nową funkcję, która służy do przeliczania wartości ze stopnii Celsjusza na stopnie Fahrenheita. Jak nazwiesz taką funkcję?
5) Stwórz nową funkcję, która służy do przeliczania wartości z mil lądowych na kilometry. Jak nazwiesz taką funkcję?
6) Stwórz nową funkcję, która służy do przeliczania wartości z metrów na sekundę na kilometry na godzinę. Jak nazwiesz taką funkcję?
7) Stwórz nową funkcję, która służy do przeliczania wartości z metrów na sekundę na mile lądowe na godzinę. Jak nazwiesz taką funkcję?
8) Stwórz nową funkcję, która służy do wyliczania pola trapezu na podstawie długości podstaw oraz wysokości trapezu. Jak nazwiesz taką funkcję?
9) Wykonaj poniższy kod. Co oznacza uzyskany wynik?

```
mean()
```

10) Wykonaj poniższy kod. Co oznacza uzyskany wynik?

```
mean("abecadło")
```

11) Wykonaj poniższy kod. Co oznacza uzyskany wynik?

```
mean(sqrt())
```

12) Wykonaj poniższy kod. Co oznacza uzyskany wynik?

```
str_length("abecadło")
```

13) Wykonaj poniższy kod. Co oznacza uzyskany wynik?

```
u = 2
z = 3 + v
v = 7
```

4. Wyrażenia warunkowe

Języki programowania opierają się o dwa podstawowe narzędzia pozwalające na sterowanie przepływem operacji. Są to wyrażenia warunkowe oraz pętle. Wyrażenia warunkowe są głównym tematem tego rozdziału, natomiast pętle oraz ich alternatywy są omówione w rozdziale 8. Celem wyrażeń warunkowych jest wykonywanie różnego zadania w zależności od danych wejściowych.

4.1. Warunki

Wyrażenie `if` opiera się o spełnienie (lub niespełnienie) danego warunku. Jeżeli dany warunek jest spełniony, kod wewnątrz wyrażenia `if()` jest wykonywany.

```
if (warunek){
  jeżeli warunek spełniony to wykonaj operację
}
```

Wyrażenie `if` oczekuje, że warunek jest wektorem logicznym o długości jeden, tj. takim który przyjmuje wartość `TRUE` lub `FALSE`. Istnieje szereg sposób uzyskania wektora logicznego w R, jednym z nich jest zastosowanie porównania wartości.

W poniższym przykładzie wyrażenie `if()` sprawdza czy wartość obiektu `temperatura` jest wyższa niż 0. W przypadku, gdy ten warunek jest spełniony (czyli jest `TRUE`), wyświetlany jest tekst `"Dodatnia"`.

```
temperatura = 5.4
if (temperatura > 0) {
  "Dodatnia"
}
#> [1] "Dodatnia"
```

W przeciwnym razie, gdy warunek nie jest spełniony (czyli ma wartość `FALSE`), kod wewnątrz warunku nie jest wykonywany.

```
temperatura = -11
if (temperatura > 0) {
  "Dodatnia"
}
```

Warunek `if` można też tworzyć w uproszczonej formie:

```
if (warunek) spelniony else niespelniony
```

4.2. Warunki zagnieżdżone

Działanie wyrażenia `if` może być połączone z dodatkowymi wyrażeniami `else if` oraz `else`. Te dwa wyrażenia wymagają najpierw wywołania wyrażenia `if()`. Jeżeli warunek w wyrażeniu `if()` jest równy `TRUE` to wykonywany jest kod w nim zawarty, a następnie obliczenie jest kończone. W przypadku, gdy wyrażenie `if()` otrzyma wartość `FALSE`, to kod w nim zawarty nie jest wykonywany, a następuje przejście do kolejnego wyrażenia, np. `else if()` w poniższym przypadku.

```
temperatura = 8.8
if (temperatura > 0) {
  "Dodatnia"
} else if (temperatura < 0) {
  "Ujemna"
} else {
  "Zero"
}
#> [1] "Dodatnia"
```

Wyrażenie `else if()` różni się od `else` tym, że wymaga ono określenia jaki warunek ma być spełniony. W przypadku `else` wyliczane są wszystkie przypadki, które nie spełniają wcześniejszych warunków.

4.3. Operatory porównania

W tabeli 4.1 można znaleźć listę podstawowych operatorów porównania. Ich celem jest sprawdzanie pewnego warunku i zwrócenie wartości `TRUE` lub `FALSE`.

Tabela 4.1.: Operatory porównania.

Operator	Wyjaśnienie
==	Równy
!=	Nie równy
%in%	Zawiera się w
>, <	Większy/Mniejszy niż
>=, <=	Większy/Mniejszy niż lub równy

Tabela 4.2.: Operatory logiczne i funkcje pomocnicze.

Operator	Wyjaśnienie
!	Negacja (nie)
&&	Koniunkcja (i)
||	Alternatywa (lub)
all	Wszystkie
any	Którykolwiek

Wyrażenie `if()` oczekuje wektora logiczego o długości jeden. Często jednak efektem porównania może być wektor o większej długości. Przykładowo, porównanie operatorem `==` daje w wyniku wektor o długości trzy, a porównanie z użyciem `%in%` skutkuje wektorem o długości jeden.

```
x = 1
y = c(1, 2, 3)
x == y
#> [1]  TRUE FALSE FALSE
x %in% y
#> [1] TRUE
```

Sterowanie tym, żeby uzyskany wynik miał oczekiwaną długość jeden może się odbywać też z pomocą operatorów logicznych i funkcji pomocniczych (tabela 4.2).

Pozwalają one na sprawdzenie czy wszystkie (`all()`) lub którykolwiek (`any()`) z elementów obiektu przyjmuje wartość `TRUE`.

```
x = 1
y = c(1, 2, 3)
all(x == y)
#> [1] FALSE
```

```
any(x == y)
#> [1] TRUE
```

Możliwe jest też łączenie bardziej złożonych zapytań używając operatora "i" (`&&`) oraz operatora "lub" (`||`).

```
x = 1
y = c(1, 2, 3)
z = 4
(x %in% y) || !(z %in% y)
#> [1] TRUE
```

Powyżej nastąpiło sprawdzenie czy element z obiektu `x` znajduje się w obiekcie `y`, a następnie czy element z obiektu `z` nie znajduje się w obiekcie `y`. Po wykonaniu ich sprawdzeń nastąpiło ich połączenie używając operatora `||`, który daje wartość `TRUE`, gdy chociaż jedno z zapytań jest prawdziwe.

W R istnieją dwa dodatkowe operatory logiczne `&` i `|`, które są zwektoryzowaną wersją operatorów `&&` i `||`. Pierwsze dwa porównują wszystkie elementy zadanych wektorów i ich wynikiem może być wektor o długości większej niż 1. Operatory `&&` i `||` porównują tylko pierwszy element każdego wektora, a w efekcie zawsze zwracają tylko jedną wartość. Dodatkowo, to one są zazwyczaj używane w wyrażeniach warunkowych.

4.4. Wyrażenia warunkowe w funkcjach

Wyrażenia warunkowe są często używanym elementem przy tworzeniu funkcji. Pozwalają one na nie tylko na określanie tego w jaki sposób dana funkcja zadziała, ale też pełnią rolę w sprawdzaniu czy do funkcji zostały wprowadzone poprawne argumenty.

Celem poniższej funkcji `pogoda()` jest wyświetlenie pewnego tekstu w zależności od podanej wartości argumentu `temperatura`. Pierwszym warunkiem, który można sprawdzić jest określenie czy użytkownik wprowadził do funkcji w postaci argumentu oczekiwany typ danych (więcej o typach danych można dowiedzieć się w rozdziale 5). W tym przypadku typ numeryczny jest oczekiwany, co można sprawdzić używając funkcji `is.numeric()`, która zwraca `TRUE` dla danych numerycznych i `FALSE` dla każdych innych.

```
pogoda = function(temperatura){
  if (is.numeric(temperatura)){
    cat(paste("Dzisiaj jest", temperatura, "stopni Celsjusza."))
  }
}
pogoda(10)
#> Dzisiaj jest 10 stopni Celsjusza.
pogoda(-20)
#> Dzisiaj jest -20 stopni Celsjusza.
pogoda("nie wiem")
```

Efekt działania powyższej funkcji jest teraz zależy od wejściowego typu danych - jeżeli podana jest wartość numeryczna zwracany jest tekst, a jeżeli ten warunek nie jest spełniony to nic się nie dzieje.

Warto, aby tworzona funkcja obsługiwała najczęściej potencjalnie używane rodzaje danych wejściowych. W tym przypadku, warto dodać wyrażenie `else`, którego efektem jest kolejny tekst sugerujący, że funkcja została wykonana, ale w inny sposób.

```
pogoda = function(temperatura){
  if (is.numeric(temperatura)){
    cat(paste("Dzisiaj jest", temperatura, "stopni Celsjusza."))
  } else {
    cat("Dzisiaj nie mamy pomiarów temperatury.")
  }
}
pogoda(10)
#> Dzisiaj jest 10 stopni Celsjusza.
pogoda(-20)
#> Dzisiaj jest -20 stopni Celsjusza.
pogoda("nie wiem")
#> Dzisiaj nie mamy pomiarów temperatury.
```

Wyrażenia warunkowe można też wielokrotnie zagnieżdżać wewnątrz zdefiniowanej funkcji.

```
pogoda = function(temperatura){
  if (is.numeric(temperatura)){
    cat(paste("Dzisiaj jest", temperatura, "stopni Celsjusza.\n"))
    if (temperatura < 5){
      cat("Ubierz się ciepło!")
```

```
    }
  } else {
    cat("Dzisiaj nie mamy pomiarów temperatury.")
  }
}
pogoda(10)
#> Dzisiaj jest 10 stopni Celsjusza.
pogoda(-20)
#> Dzisiaj jest -20 stopni Celsjusza.
#> Ubierz się ciepło!
pogoda("nie wiem")
#> Dzisiaj nie mamy pomiarów temperatury.
```

Przykładowo, powyżej komunikat `"Ubierz się ciepło!"` jest wyświetlany w momencie, gdy spełnione zostaną dwa warunki - najpierw wejściowy obiekt `temperatura` musi być typu numerycznego, a następnie wartość tego obiektu musi być niższa niż 5.

4.5. Zadania

1) Spójrz na poniższe przykłady, ale ich nie wykonuj. Co będzie wynikiem działania każdego z tych przykładów?

```
liczby = c(1, 2)
liczby == 1          #1
liczby != 1          #2
liczby %in% 1        #3
all(liczby %in% 1)   #4
any(liczby %in% 1)   #5
```

2) Spójrz na cztery poniższe przykłady, ale ich nie wykonuj. Co będzie wynikiem działania każdego z tych przykładów?

```
(c(1, 2) > 0) &  (c(-1, 2) > 0) #1
(c(1, 2) > 0) && (c(-1, 2) > 0) #2
(c(1, 2) > 0) |  (c(-1, 2) > 0) #3
(c(1, 2) > 0) || (c(-1, 2) > 0) #4
```

3) Napisz funkcję, która przyjmuje trzy zmienne logiczne `x`, `y` i `z`. Jeżeli tylko jedna lub trzy ze zmiennych ma wartość `TRUE` wyświetl tekst `"Nieparzysta liczba."`, natomiast jeżeli dwie zmienne mają wartość `TRUE` wyświetl tekst `"Parzysta liczba."`

4) Napisz funkcję, która przyjmuje dwie zmienne numeryczne `x` i `y`. Jeżeli wszystkie wartości zmiennej `x` są większe od `y` wyświetl tekst `"Zwycięstwo."`, a w przeciwnym razie wyświetl tekst `"Porażka."`.
5) Napisz funkcję, która przyjmuje dwie zmienne numeryczne `populacja` i `powierzchnia`. Jeżeli wartości gęstości zaludnienia (liczba osób na jednostkę powierchni) jest wyższa niż 123 wyświetl tekst `"Wartość powyżej średniej dla Polski."`.

5. Proste obiekty

Obiekty w R można podzielić na proste (homogeniczne) i złożone (heterogeniczne). Do podstawowych prostych obiektów należą wektory atomowe (ang. *vector*) i macierze (ang. *matrix*), natomiast listy (ang. *list*) i ramki danych (ang. *data frame*) to obiekty złożone.

W tym rozdziale skupimy się na wektorach atomowych, dla uproszczenia nazywanych dalej po prostu wektorami. Pozostałe podstawowe typy obiektów są omówione w rozdziale 7. Więcej informacji na temat podstawowych typów obiektów można znaleźć w rozdziale "Vectors"[1] książki Advanced R (Wickham, 2014).

5.1. Wektory

Wektory są podstawowymi elementami, które pozwalają na budowanie bardziej złożonych rodzajów obiektów. Wektor może przyjmować jeden z czterech podstawowych typów[2]:

1. logiczny (ang. *logical*)

```
wek_log = c(TRUE, FALSE)
wek_log
#> [1]  TRUE FALSE
```

2. liczba całkowita (ang. *interger*)

```
wek_cal = c(5L, -7L)
wek_cal
#> [1]  5 -7
```

3. liczba zmiennoprzecinkowa (ang. *double*)[3]

[1] https://adv-r.hadley.nz/vectors-chap.html

[2] Istnieją też dwa kolejne podstawowe typy wektorów, złożone (ang. *complex*) oraz surowe (ang. *raw*) ale są one bardzo rzadko używane.

[3] Liczby zmiennoprzecinkowe mogą być też reprezentowane poprzez notację naukową, np. 11111.2 może być zapisane jako 1.11112e4, a 0.00021 jako 2.1e-4.

```
wek_zmi = c(5.3, -7.1)
wek_zmi
#> [1]  5.3 -7.1
```

4. znakowy (ang. *character*)

```
wek_zna = c("kot", "pies")
wek_zna
#> [1] "kot"  "pies"
```

Wektory przedstawiające liczby stałoprzecinkowe i zmiennoprzecinkowe są często łączone i wspólnie określane jako wektory numeryczne (ang. *numeric*).

Wiele języków programowania posiada zmienne skalarne (tzw. skalary), czyli takie które mogą przyjmować tylko jedną wartość. W R one nie występują, zamiast nich stosowane są wektory o długości jeden.

Dodatkowo, istnieje wiele dodatkowych, rzadziej spotykane typów wektorów - czynnikowy (ang. *factor*), dat (ang. *date*) i czasu (ang. *date-time*) (sekcje 5.11, 5.12 i 5.13).

5.2. Właściwości wektorów

Każdy wektor ma trzy właściwości - typ, długość i atrybuty. Typ może być sprawdzony używając funkcji `typeof()`.

```
# typ
typeof(wek_zmi)
#> [1] "double"
```

Celem funkcji `length()` jest sprawdzenie długości wektora, czyli tego z ilu wartości (elementów) się on składa.

```
# długość
length(wek_zmi)
#> [1] 2
```

Atrybuty pozwalają na dodawanie nowych informacji do wektorów atomowych, a w efekcie dają też możliwość tworzenia bardziej złożonych struktur (rozdział 7).

```
# atrybuty
attributes(wek_zmi)
#> NULL
```

5.3. Podstawowe funkcje

Z racji bycia podstawowym typem obiektu w R, wektory są używane w bardzo dużej liczbie funkcji. Kilka z podstawowych, często przydatnych funkcji jest podana i wyjaśniona poniżej.

Funkcja `str()` ma na celu wyświetlenie **str**uktury danych. W przypadku wektorów oznacza to skrót od nazwy typu danych (`logi` - logiczny, `int` - stałoprzecinkowy, `num` - zmiennoprzecinkowy (numeryczny), `chr` - tekstowy), jego długość (np. `[1:2]` oznacza, że wektor ma dwa elementy), oraz kilka przykładowych wartości tego wektora.

```
str(wek_cal)
#>  int [1:2] 5 -7
```

Funkcja `names()` wyświetla nazwy przypisane kolejnym elementom wektora.

```
names(wek_zna)
#> NULL
```

W powyższym przypadku wektor `wek_zna` nie miał żadnych nazw, w efekcie funkcja `names()` zwróciła `NULL` (więcej informacji na temat `NULL` można znaleźć w sekcji 5.5). Oprócz wyświetlania nazw, funkcja `names()` daje też możliwość ich nadania.

```
names(wek_zna) = c("a", "b")
wek_zna
#>      a      b
#>  "kot" "pies"
names(wek_zna)
#> [1] "a" "b"
```

Funkcja `seq` ma na celu generowanie ciągów liczbowych[4]. Pierwszym jego argumentem jest `from` czyli początkowa liczba w ciągu a drugi argument `to` oznacza maksymalną możliwą liczbę w ciągu. Obie te liczby mogą być wektorami o długości jeden. Dodatkowo ta funkcja wymaga zdefiniowania jeszcze jednego argumentu, np. `by` lub `length.out`. Argument `by` określa co ile wartości w ciągu mają rosnąć od wartości początkowej.

```
seq(1, 365, by = 7)
#>  [1]   1   8  15  22  29  36  43  50  57  64  71  78  85
#> [14]  92  99 106 113 120 127 134 141 148 155 162 169 176
#> [27] 183 190 197 204 211 218 225 232 239 246 253 260 267
#> [40] 274 281 288 295 302 309 316 323 330 337 344 351 358
#> [53] 365
```

Alternatywnie, argument `length.out` ustala jakiej długości ma być wynikowy ciąg, a na podstawie tego tworzone są wartości w równych odstępach.

```
seq(1, 365, length.out = 10)
#>  [1]   1.0  41.4  81.9 122.3 162.8 203.2 243.7 284.1
#>  [9] 324.6 365.0
```

Funkcja `rep` służy powielaniu zadanej wartości podaną liczbę razy. W poniższym przykładzie, wartość 11 jest powielona 4 razy.

```
rep(11, 4)
#> [1] 11 11 11 11
```

Ta funkcja działa też na różnego typu wektorach - logicznych, numerycznych, czy tekstowych.

```
rep(wek_zna, 4)
#>      a      b      a      b      a      b      a      b
#>  "kot" "pies"  "kot" "pies"  "kot" "pies"  "kot" "pies"
```

[4]Uproszczeniem funkcji `seq` jest operator `:` (np. `1:10`). W jego przypadku wartości zawsze zmieniają się o jeden.

5.4. Działania na wektorach

Wiele podstawowych operacji w R jest zwektoryzowana. Przykładowo, możliwe jest pomnożenie kolejnych elementów jednego wektora przez kolejne elementy drugiego wektora.

```
a = c(1, 2, 3)
b = c(3, 5, 10)
a * b
#> [1]  3 10 30
```

Kod zapisany w powyższy sposób zajmuje niewiele miejsca i jest łatwy do odczytania. Alternatywnie można by ten problem rozbić na podelementy i je wymnożyć.

```
a1 = 1
a2 = 2
a3 = 3
b1 = 3
b2 = 5
b3 = 10
a1 * b1
#> [1] 3
a2 * b2
#> [1] 10
a3 * b3
#> [1] 30
```

Jak można szybko zaobserwować, mnożenie kolejnych elementów w ten sposób wymaga zapisania znacznie więcej kodu i jest trudniejsze do odczytania. Jest jeszcze trzecia możliwość użycie pętli (rozdział 8).

```
y = numeric(length = 3)
for (i in 1:3){
  y[i] = a[i] * b[i]
}
y
#> [1]  3 10 30
```

Stworzony kod zajmuje mniej miejsca, ale nadal nie jest on bardzo łatwy do szybkiego zrozumienia.

Wektoryzacja ma też inną zaletę - obliczenia wykonywane w ten sposób są szybkie. W przypadku stosowania niektórych operacji, np. mnożenia czy dodawania, R wykorzystuje w tle (bez wiedzy użytkownika) zoptymalizowane funkcje zapisane w języku C lub Fortran.

W przypadku, gdy dwa wektory mają różną długość, wówczas następuje proces nazwany recyklingiem (ang. *recycling*) - elementy krótszego wektora są powtarzane aż do momentu gdy osiągnie on taką samą długość jak ten dłuższy, a dopiero później następuje wykonanie wybranego działania. W takiej sytuacji pojawi się też poniższy komunikat ostrzeżenia.

```
a = c(1, 2, 3)
d = c(3, 5)
a * d
#> Warning in a * d: longer object length is not a multiple
#> of shorter object length
#> [1]  3 10  9
```

Więcej informacji na temat wektoryzowania kodu można znaleźć w rozdziale 8.

5.5. Brakujące wartości

Wyobraź sobie, że wykonujesz codziennie o 12:00 pomiar temperatury.

```
temperatura = c(8.2, 10.3, 12.0)
```

Czwartego dnia twój termometr się popsuł i nie można było wykonać pomiaru. Co należałoby w takim razie zrobić? Można by pominąć ten pomiar, naprawić termometr i wykonać pomiar kolejnego dnia. Wówczas jednak mielibyśmy cztery wartości dla pięciu dni. Inną możliwą opcją byłoby użycie wartości, która stałaby się kodem wartości brakujących, np. 999. Problemem tego rozwiązania jest to w jaki sposób należałoby, np. wyliczyć średnią w tym obiekcie.

```
temperatura = c(8.2, 10.3, 12.0, 999)
```

Najlepszą opcją byłoby wykorzystanie wbudowanego oznaczenia wartości brakujących w R - NA.

```
temperatura = c(8.2, 10.3, 12.0, NA)
```

Zachowanie wartości NA (ang. *Not Available*) jest bardzo intuicyjne. Przykładowo, jeżeli nie znamy jakiejś wartości to jeżeli dodamy do niej 2 to również nie wiemy jaki mamy wynik.

```
NA + 2
#> [1] NA
5 > NA
#> [1] NA
```

Podobnie będzie w sytuacji, gdy chcemy wyliczyć średnią na podstawie wektora, który zawiera wartość NA.

```
mean(temperatura)
#> [1] NA
```

W takich przypadkach najpierw należałoby usunąć wartość NA a następnie wyliczyć średnią z pozostałych wartości w tym wektorze. Aby ułatwić taką operację w wielu funkcjach istnieje argument na.rm. W momencie, gdy jest on ustalony na TRUE, to wszystkie przypadki NA są usuwane na potrzeby wyliczania średniej.

```
mean(temperatura, na.rm = TRUE)
#> [1] 10.2
```

Do sprawdzenia czy w wektorze znajduje się wartość NA służy funkcja is.na().

```
is.na(temperatura)
#> [1] FALSE FALSE FALSE  TRUE
```

R posiada też kilka dodatkowych specjalnych obiektów, takich jak NULL, NaN, Inf oraz -Inf. NULL ma długość zero i nie posiada żadnych atrybutów. Może on posłużyć np. do usuwania kolumn w ramkach danych. NaN (ang. *Not a Number*) oznacza wartość, która nie jest zdefiniowaną lub nie może być reprezentowana w inny sposób, przykładowo 0/0. Inf i -Inf (ang. *Infinity*) jest wynikiem obliczeń, które dały bardzo dużą wartość dodatną lub ujemną, przykładowo 9^999.

5.6. Wydzielanie

R posiada trzy podstawowe operatory wydzielania (ang. *subsetting*) - `[]`, `[[]]` oraz `$`, które działają w różny sposób w zależności od tego czy wydzielamy wektory, macierze, ramki danych czy listy. W tym rozdziale skupimy się na wydzielaniu elementów z wektora przy użyciu operatora `[]`. Więcej na temat wydzielania innych obiektów można znaleźć w rozdziale 7.

Wydzielanie wektorów używając operatora `[]` może odbywać się używając jednego z poniższych zapytań:

1. Na podstawie pozycji.
2. Na podstawie wektora logicznego.
3. Na podstawie nazwy.
4. Używając elementu pustego.
5. Używając zera.

W przypadku wydzielania na podstawie pozycji konieczne jest podanie wektora, który wskazuje numer elementów, które mają zostać wybrane.

```
temperatura[c(1, 3)]
#> [1]  8.2 12.0
```

Alternatywnie, możliwe jest też użycie znaku minus (-) przed definicją pozycji. Wówczas wybrane zostaną wszystkie elementy wektora oprócz tych w podanych pozycjach.

```
temperatura[-c(2, 4)]
#> [1]  8.2 12.0
```

Drugim sposobem jest użycie wektora logicznego. W takim przypadku elementy określone jako prawda (`TRUE`) zostają wybrane.

```
temperatura[c(TRUE, FALSE, TRUE, FALSE)]
#> [1]  8.2 12.0
```

Możliwe jest też stworzenie wektora logicznego poprzez wykonanie prostego zapytania. Poniżej wybrano tylko te elementy wektora `temperatura`, gdzie wartość w wektorze `temperatura` była wyższa niż 10.

```
temperatura[temperatura > 10]
#> [1] 10.3 12.0   NA
```

Wydzielanie na podstawie nazwy wymaga aby elementy w wektorze były nazwane.

```
names(temperatura) = c("Poniedziałek", "Wtorek", "Środa", "Czwartek")
temperatura
#> Poniedziałek       Wtorek        Środa     Czwartek
#>          8.2         10.3         12.0           NA
```

Następnie wybór elementów odbywa się poprzez podanie nazw wybranych elementów.

```
temperatura[c("Wtorek", "Czwartek")]
#>   Wtorek Czwartek
#>     10.3       NA
```

Czwartą metodą jest zapytanie w oparciu o pusty element. W przypadku prostych obiektów, efektem takiego wydzielania jest oryginalny wektor. Ta metoda jest jednak bardzo użyteczna dla obiektów złożonych, takich jak macierze czy ramki danych (rozdział 7).

```
temperatura[]
#> Poniedziałek       Wtorek        Środa     Czwartek
#>          8.2         10.3         12.0           NA
```

Ostatnią metodą jest użycie zera. W efekcie zostanie zwrócony wektor o długości zero, ale zachowujący oryginalne właściwości, jak np. bycie wektorem numerycznym (`numeric`) poniżej. Takie zachowanie może być przydatne podczas tworzenia nowych obiektów w oparciu o już istniejące.

```
temperatura[0]
#> named numeric(0)
```

5.7. Wydzielanie i przypisanie

Wydzielanie elementów może mieć kilka dodatkowych zastosowań oprócz wyświetlania wybranych wartości. Kolejną możliwością jest tworzenie nowych obiektów na podstawie wydzieleń. W takich sytuacjach działa każda z metod wyjaśnionych w powyższej sekcji, np. wydzielenie przez nazwę.

```
temperatura_pon = temperatura["Poniedziałek"]
temperatura_pon
#> Poniedziałek
#>          8.2
```

Dodatkowo, wydzielanie może przyjmować bardziej złożoną postać. Poniżej nastąpiło stworzenie nowego obiektu `temperatura_10` składającego się z wszystkich elementów wektora `temperatura`, których wartość jest wyższa od 10 i nie jest `NA`.

```
temperatura_10 = temperatura[temperatura > 10 & !is.na(temperatura)]
temperatura_10
#> Wtorek  Środa
#>   10.3   12.0
```

5.8. Modyfikowanie obiektów

Trzecim zastosowaniem wydzielania jest modyfikowanie obiektów. Poprzez wydzielenie możliwe jest wskazanie, które elementy wektora mają być zamienione i na jakie wartości. Przykładowo, wektor `temperatura` zawiera cztery wartości nazwane kolejnymi dniami tygodnia. W tym wektorze `"Czwartek"` posiada wartość brakującą `NA`.

```
temperatura
#> Poniedziałek       Wtorek        Środa     Czwartek
#>          8.2         10.3         12.0           NA
temperatura["Czwartek"]
#> Czwartek
#>       NA
```

Aby zamienić wartość tego elementu należy go wydzielić a następnie przypisać temu elementowi nową wartość. Efektem jest trwała zmiana obiektu `temperatura`.

```
temperatura["Czwartek"] = 9.1
temperatura["Czwartek"]
#> Czwartek
#>      9.1
temperatura
#> Poniedziałek       Wtorek        Środa     Czwartek
#>          8.2         10.3         12.0          9.1
```

5.9. Łączenie podstawowych typów obiektów

Właściwością wektora jest to, że może on przyjmować tylko jeden typ. Przykładowo poniżej wyświetlony jest wektor logiczny przyjmujący wartość `FALSE`.

```
c(FALSE)
#> [1] FALSE
```

Co stanie się, gdy będziemy chcieli taki wektor połączyć z wektorem innego typu, np. numerycznego czy tekstowego? Próba stworzenia obiektu składającego się z wielu typów spowoduje wymuszenie (ang. *coercion*) do najbliższego możliwego typu. Odbywa się to zgodnie z zasadą: logiczny -> liczba całkowita -> liczba zmiennoprzecinkowa -> znakowy.

Łącząc wektor logiczny i liczby całkowitej otrzyma się wektor składający się z liczb całkowitych, gdzie `FALSE` zostanie zamienione na `0`.

```
c(FALSE, 2L)
#> [1] 0 2
```

Łączenie wektorów logicznego, liczby całkowitej i liczby zmiennoprzecinkowej w efekcie da wektor zmiennoprzecinkowy.

```
c(FALSE, 2L, 3.1)
#> [1] 0.0 2.0 3.1
```

W sytuacji, gdy którykolwiek element będzie tekstem, cały wektor zostaje zamieniony na tekst.

```
c(FALSE, 2L, 3.1, "kot")
#> [1] "FALSE" "2"     "3.1"   "kot"
```

5.10. Zmiana typów obiektów

Do zmiany typu obiektu[5] służą funkcje `as.logical()`, `as.integer()`, `as.double()`, oraz `as.character()`.

```
as.logical(c("FALSE", "TRUE")) # znakowy na logiczny
#> [1] FALSE  TRUE
as.integer(c("3", "2")) # znakowy na liczba całkowita
#> [1] 3 2
as.double(c(3L, 2L)) # liczba całkowita na liczba zmiennoprzecinkowa
#> [1] 3 2
as.character(c(3L, 2L)) # liczba całkowita na znakowy
#> [1] "3" "2"
```

Do sprawdzenia czy dany obiekt należy do wybranego typu służą funkcje `is.logical()`, `is.integer()`,`is.double()`, oraz `is.character()`.

5.11. Wektory czynnikowe

Wektory czynnikowe służą do przechowywania informacji o pewnych kategoriach. Mogą to być, na przykład, wielokrotnie powtórzone nazwy miast czy krajów, czy też określenia płci w danych statystycznych. Wektory czynnikowe wspomagają określanie wartości dla kolejnych grup o tej samej nazwie.

```
tekst = c("Poznań", "Kraków", "Warszawa", "Poznań")
```

Zamiana wektora tekstowego na czynnikowy odbywa się z użyciem funkcji `as.factor()`.

```
czynn = as.factor(tekst)
czynn
#> [1] Poznań   Kraków   Warszawa Poznań
#> Levels: Kraków Poznań Warszawa
```

[5] Taka operacja często jest określana jako rzutowanie.

Wektory czynnikowe są wewnętrznie w R reprezentowane jako wartości stałoprzecinkowe. Dodatkowo, posiadają one pewne informacje zaszyte w atrybutach, w tym wartości wszystkich kategorii oraz stwierdzenie posiadanej klasy [^Posiadanie attubutu `class` zamienia je w tak zwane obiekty S3, które zachowują się inaczej niż normalne wektory atomowe.].

```
typeof(czynn)
#> [1] "integer"
length(czynn)
#> [1] 4
attributes(czynn)
#> $levels
#> [1] "Kraków"   "Poznań"   "Warszawa"
#>
#> $class
#> [1] "factor"
```

Możliwa jest również zamiana w drugą stronę - z wektora czynnikowego na wektor tekstowy używając funkcji `as.character()`.

```
tekst2 = as.character(czynn)
tekst2
#> [1] "Poznań"   "Kraków"   "Warszawa" "Poznań"
```

5.12. Wektory dat

R ma wbudowaną reprezentację dat w postaci klasy `Date`.

```
dzis = Sys.Date()
dzis
#> [1] "2019-06-01"
```

Pomimo tego, że powyżej data jest wyświetlona jako tekst (zwróć uwagę na cudzysłowia), wewnętrznie w R jest ona reprezentowana jako wartość zmiennoprzecinkowa.

```
typeof(dzis)
#> [1] "double"
length(dzis)
#> [1] 1
attributes(dzis)
#> $class
#> [1] "Date"
```

Sprawdzenie tej wartości możliwe jest poprzez użycie funkcji `unclass()`.

```
unclass(dzis)
#> [1] 18048
```

Wynik, 18048, oznacza liczbę dni od 1970-01-01.[6] W tej reprezentacji dni przed 1970-01-01 określane wewnętrznie są poprzez wartości ujemne.

```
stara_data = as.Date("1912-04-13")
unclass(stara_data)
#> [1] -21082
```

Tworzenie wektora dat odbywa się używając funkcji `as.Date()`.

```
daty = as.Date(c("2011-02-02", "2011-02-03"))
daty
#> [1] "2011-02-02" "2011-02-03"
```

Funkcja `as.Date()` oczekuje podanych wartość w postaci *YYYY-MM-DD* (wyjaśnienie można znaleźć w sekcji 2.4.5), ale możliwe jest również wymuszenie innej postaci danych wejściowych poprzez użycie argumentu `format`.

5.13. Wektory czasu

W R istnieją również wbudowane reprezentacje dat i godzin (inaczej zwane data-czas, ang. *date-times*). Najczęściej używaną jest klasa `POSIXct`, która jest wektorem przedstawiającym liczbę sekund of 1970-01-01.[7]

[6] https://en.wikipedia.org/wiki/Unix_time
[7] Wyświetlenie aktualnego czasu jest możliwe używając funkcji `Sys.time()`.

```
czas = as.POSIXct("2011-02-02 10:33", tz = "CET")
czas
#> [1] "2011-02-02 10:33:00 CET"
```

Wewnętrznie w R jest ona również reprezentowana jako wartość zmiennoprzecinkowa.

```
typeof(czas)
#> [1] "double"
length(czas)
#> [1] 1
attributes(czas)
#> $class
#> [1] "POSIXct" "POSIXt"
#>
#> $tzone
#> [1] "CET"
```

Przykładowo, `"2011-02-02 10:33"` miało miejsce 1296639180 sekund od 1970-01-01.

```
unclass(czas)
#> [1] 1.3e+09
#> attr(,"tzone")
#> [1] "CET"
```

Ważnym elementem reprezentacji czasu jest określenie strefy czasowej. Można ją zdefiniować w funkcji `as.POSIXct()` używając argumentu `tz`, ale też można zmienić strefę czasową istniejącego wektora poprzez modyfikację jego atrybutów. W poniższym przykładzie nastąpiła zmiana strefy czasowej z czasu środkowoeuropejskiego (`"CET"`) na czas pacyficzny [8].

```
attributes(czas)$tzone = "America/Los_Angeles"
czas
#> [1] "2011-02-02 01:33:00 PST"
```

Więcej informacji na temat stref czasowych używanych w R można znaleźć w pliku pomocy `?timezones`.

[8] https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

R posiada też dodatkowe klasy specjalne, np `POSIXlt` przechowująca informacje o dacie w postaci listy oraz `difftime` reprezentująca czas trwania.

5.14. Zadania

1) Wykonujesz trzy razy dziennie (o godzinie 7:00, 15:00 i 23:00) pomiar temperatury. W ostatnich dwóch dniach (2019-03-11 i 2019-03-12) pomierzone wartości to 0, 5,2, 3,9, 4,1, 8,7, 5,3 stopni Celsjusza. Stwórz nowy wektor `pomiary` zawierający te wartości.
2) Nazwij kolejne elementy tego wektora używając kolejnych liter alfabetu.
3) Pomyśl jakie właściwości ma wektor `pomiary` - jaki ma typ, jaką ma długość, jakie ma atrybuty? Jeżeli jesteś przekonany co do odpowiedzi to sprawdź to pisząc odpowiedni kod.
4) Znajdź i wyświetl wartość `4.1` w wektorze `pomiary` używając kilku różnych sposobów wydzielania. Ile możliwości udało się Tobie znaleźć?
5) Znajdź automatycznie położenie najwyższej wartości wektora `pomiary` i na jej podstawie stwórz nowy obiekt `pomiary_max`.
6) Wydziel wszystkie pomiary równe i niższe niż 5 stopni Celsjusza. Na ich podstawie stwórz nowy obiekt `pomiary_n5`.
7) Inna osoba również wykonywała pomiary temperatury w tym samym czasie. Jej pomierzone wartości to -1.1, 4,2, 2,4, 3,1, 7,1, 4,2 stopni Celsjusza. Dodaj te wartości do obiektu `pomiary`.
8) Stwórz nowy wektor `nazwy_stacji` zawierający nazwę Twojej stacji pomiarowej (`"Punkt 31"`) oraz stacji drugiej osoby (`"Stacja Thule"`), którego długość ma być równa długości wektora `pomiary`. Wektor `nazwy_stacji` powinien być klasy czynnikowej.
9) Stwórz nowy wektor `daty_pomiarow` zawierający rok, miesiąc i dzień pomiarów w wektorze `pomiary`.
10) Stwórz nowy wektor `czas_pomiarow` zawierający datę i czas pomiarów w wektorze `pomiary`. Zdefiniuj też odpowiednią strefę czasową.
11) Stwórz nowy wektor `id_pomiarow` zawierający kolejne liczby całkowite od 1 do liczby pomiarów w wektorze `pomiary`.
12) Zastanów się (bez wykonywania) co jest efektem działania poniższego kodu. Kiedy taka operacja mogłaby być konieczna?

```
pomiary[3] = 3.3
```

13) Zastanów się (bez wykonywania) co jest efektem działania poniższego kodu. Kiedy taka operacja mogłaby być konieczna?

```
pomiary[pomiary <= 0] = NA
```

14) Zastanów się (bez wykonywania) co jest efektem działania poniższego kodu.

```
wartosci1 = as.character(c(1, 3, 5))
mean(wartosci1)
```

15) Zastanów się (bez wykonywania) co jest efektem działania poniższego kodu.

```
wartosci2 = as.numeric(c(1, "trzy", 5))
wartosci2
mean(wartosci2)
```

16) Wykonaj poniższą funkcję. Następnie wydziel nowy wektor `pomiary2`, który nie zawiera wartości `NA`.

```
pomiary[pomiary <= 0] = NA
```

17) Trzecia osoba również wykonywała pomiary temperatury w tym samym czasie. Przesłała ona Tobie taki wektor - `c(-5.2, 3.0, 1.1, "zaspałem", 6.4, 2.2)`. Jakiej klasy jest ten wektor a jaka powinna być jego klasa, aby możliwe było wykonywanie na niej obliczeń, np. wyliczanie średniej? Jak to można uzyskać?

6. Tekst

Podstawowymi typami danych w R są wektory logiczne, numeryczne i znakowe (sekcja 5.1). Pierwsze z nich przyjmują dwie formy - `TRUE` i `FALSE`, przez co istnieje pewna skończona liczba operacji, które można na nich wykonać. Wektory numeryczne mogą przyjmować wiele form, ale najczęściej są one przetwarzanie używając podstawowych operacji arytmetycznych, takich jak dodawanie, odejmowanie, mnożenie czy dzielenie. Wektory znakowe są natomiast najbardziej zróżnicowane - mogę przyjmować różne formy (nawet w zależności od przyjętego alfabetu), w tym pozwalają one także na przechowywanie wartości logicznych czy numerycznych.

Celem tego rozdziału jest przedstawienie najczęściej spotykanych operacji na tekście, takich jak jego wyszukiwanie, wydzielanie czy zamiana. Więcej na temat przetwarzania tekstu można znaleźć w rozdziale "Strings"[1] książki R for Data Science (Wickham and Grolemund, 2016).

6.1. Reprezentacja tekstu

Typ znakowy jest określany poprzez użycie cudzysłowia `"` lub `'`. Ważne tutaj jest, aby rozpoczynać i kończyć tekst tym samym cudzysłowiem.

```
t1 = "kot"
t2 = 'pies'
t3 = '"W teorii, teoria i praktyka są tym samym. W praktyce, nie są." - Yogi Berra'
```

W momencie, gdy tekst nie będzie kończył się cudzysłowiem, wykonanie kodu jest niemożliwe. Wówczas zamiast znaku `>`, oznaczającego nową linię wykonywanego kodu, pojawi się znak `+`. Oznacza on, że wykonanie kodu nie może zostać zakończone.

```
> "Mój pierwszy alfabet
+
+
```

W takiej sytuacji należy nacisnąć klawisz Esc, aby przerwać wykonywanie operacji, a następnie poprawić wpisany kod.

[1] https://r4ds.had.co.nz/strings.html

6.2. Podstawowe operacje na tekście

Jedną z podstawowych operacji na wektorach znakowych jest ich łączenie. Do tego celu służy funkcja `paste()`[2].

```
paste("t", "o", " ", "k", "o", "t")
#> [1] "t o   k o t"
```

Efekt działania funkcji `paste()` jest jeden wektor tekstowy, który składa się z wejściowych wektorów oddzielonych domyślnie spacjami. Funkcja `paste()` ma jednak również dodatkowy argument `sep`, który pozwala na dowolne określanie separatora. Ostatnim argumentem tej funkcji jest `collapse`, który łączy elementy jednego wektora tekstowego.

R oferuje też uproszczoną postać tej funkcji o nazwie `paste0()`, w której nie ma znaku seperatora.

```
paste0("t", "o", " ", "k", "o", "t")
#> [1] "to kot"
```

Te funkcje są używane w sytuacjach, gdy chcemy połączyć stały, znany tekst, wraz z tekstem wprowadzanym przez użytkownika lub pochodzącym z innego źródła. Poniżej stworzono dwie nowe zmienne `imie` i `wiek`, których treść złączono ze słowami `"ma"` i `"lat."`.

```
imie = "Olek"
wiek = 77
tekst1 = paste(imie, "ma", wiek, "lat.")
tekst1
#> [1] "Olek ma 77 lat."
```

Takie konstrukcje są często używane w funkcjach. Powyższy przykład możnaby przepisać jako:

```
lata = function(imie, wiek){
  paste(imie, "ma", wiek, "lat.")
}
lata("Asia", 61)
#> [1] "Asia ma 61 lat."
```

[2]Odpowiednikiem funkcji `paste()` w pakiecie **stringr** jest funkcja `str_c()`

Dodatkowo w R istnieje alternatywa dla `paste()` i `paste0()` w postaci funkcji `sprintf()`.

Kolejne podstawowe funkcje, `toupper()` i `tolower()` zamieniają cały istniejący tekst na taki który posiada tylko duże lub małe litery.

```
toupper(tekst1)
#> [1] "OLEK MA 77 LAT."
tolower(tekst1)
#> [1] "olek ma 77 lat."
```

Są one używane w sytuacjach, gdy posiadamy dane, w których jeden tekst jest podany w kilku formach i chcemy je ujednolicić.

R posiada też wiele innych wbudowanych funkcji do obsługi tekstu (np. `grep()`), ale istnieją też specjalne pakiety poświęcone temu zagadnieniu, w tym pakiet **stringr** (Wickham, 2019a).

```
library(stringr)
```

Większość funkcji tego pakietu zaczyna się od prefiksu `str_` co ułatwia znajdowanie funkcji w tym pakiecie i zmniejsza szansę na nałożenie się funkcji o takiej samej nazwie z innego pakietu.

Przykładową operacją na tekście jest jego sortowanie (czyli układanie alfabetycznie), do którego służy funkcja `str_sort()`.

```
tekst2 = c("czosnek", " hałas", "ćma ")
tekst2
#> [1] "czosnek" " hałas"  "ćma "
str_sort(tekst2)
#> [1] " hałas"  "ćma "    "czosnek"
```

W powyższym przykładzie oczkiwalibyśmy ułożenia, w których `"hałas"` byłby na ostatnim miejscu. Nie jest tak z powodu istnienia z przodu tego wyrazu znaku niedrukowalnego - spacji. Aby usunąć spacje z przodu i tyłu tekstu można użyć funkcji `str_trim()`.

```
tekst2 = str_trim(tekst2)
tekst2
#> [1] "czosnek" "hałas"   "ćma"
```

W tej chwili możmy użyć funkcji `str_sort()` jeszcze raz.

```
str_sort(tekst2)
#> [1] "ćma"     "czosnek" "hałas"
```

Teraz `"hałas"` jest poprawnie na ostanim miejscu, ale na pierwszej pozycji jest `"ćma"` zamiast `"czosnek"`. Różne alfabety na świecie mają inne znaki oraz ich kolejność. Domyślnie funkcja `str_sort()` używa alfabetu angielskiego, co w efekcie powoduje niepoprawne ułożenie polskich znaków. Do rozwiązania tego problemu służy argument `locale`, w którym można określić jaki alfabet ma być używany.

```
str_sort(tekst2, locale = "pl")
#> [1] "czosnek" "ćma"     "hałas"
str_sort(tekst2, locale = "cs")
#> [1] "ćma"     "czosnek" "hałas"
```

Powyżej można zobaczyć dwa przykłady - ułożenia tekstu według polskiego i czeskiego alfabetu[3].

6.3. Wydzielanie tekstu

Częstym przypadkiem jest potrzeba wydzielenia tylko fragmentu tekstu. W tej sekcji zostanie pokazane jak wydzielać tekst na podstawie pozycji, ale możliwe jest również wydzielanie tekstu na podstawie wzorca (zobacz sekcję 6.5). Wydzielanie na podstawie pozycji jest używane w sytuacjach, gdy struktura wejściowego tekstu jest nam znana i stabilna, np. gdy interesuje nas wybranie fragmentu tekstu z automatycznie generowanych raportów.

```
tekst1 = "Olek ma 77 lat."
```

W przypadku wydzielania tekstu na podstawie pozycji należy określić pozycję pierwszego i ostatniego znaku, który nas interesuje. Można to zrobić na kilka sposobów. Pierwszy z nich polega na określeniu położenia znaków od lewej strony, np. poniższy kod wybiera tekst rozpoczynający się od 9 znaku i kończący się na znaku 15 włącznie.

[3] https://en.wikipedia.org/wiki/Czech_orthography

```
str_sub(tekst1, start = 9, end = 15)
#> [1] "77 lat."
```

Definiowanie pozycji może się też odbywać od prawej strony tekstu używając znaku -.

```
str_sub(tekst1, start = 9, end = -1)
#> [1] "77 lat."
```

W powyższym przykładzie wybierany jest tekst zaczynający się na 9 znaku a kończący na pierwszym znaku od końca włącznie. Natomiat poniżej wybrany jest tekst zaczynający się na siódmym znaku od końca i kończący na pierwszym od kónca włącznie.

```
str_sub(tekst1, start = -7, end = -1)
#> [1] "77 lat."
```

6.4. Wyrażenia regularne

Sprawdzanie czy dany tekst występuje w wektorze można wykonać używając funkcji `str_detect()`.

```
tekst3 = c("Magdalena", "Lena", "1Lena.csv", "LLena", "Helena", "Anna", "99")
```

W takim wypadku konieczne jest zdefiniowanie argumentu `pattern`, czyli wzorca tekstowego, który nas interesuje. Aby znaleźć wszystkie wystąpienia (nawet fragmentaryczne) słowa `"Lena"` można użyć poniższego kodu.

```
str_detect(tekst3, pattern = "Lena")
#> [1] FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

Jego efektem będzie wektor logiczny wskazujący, które elemeny zawierają wybrany wzorzec (`TRUE`) oraz które go nie zawierają (`FALSE`). Wzorzec zdefiniowany w tej sposób jest czuły na wielkość znaków dlatego też zapytanie używając `"Lena"` da inny wynik niż takie używając `"lena"`.

```
str_detect(tekst3, pattern = "lena")
#> [1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE
```

W celu ułatwienia wyszukiwania złożonych fraz powstały wyrażenia regularne. Wyrażenia regularne (ang. *regular expressions*), często określane jako *regex* to sposób opisywanie wzorców tekstu. Używając wyrażeń regularnych możliwe jest, między innymi, znajdowanie tekstu lub zamienienie, który spełnia wymagane warunki. Wyrażenia regularne są powszechnie używane w wyszukiwarkach internetowych, edytorach tekstu, oraz wielu językach programowania.

Wyrażenia regularne opierają się o stosowanie szeregu operatorów (metaznaków) wymienionych w tabeli 6.1.

Tabela 6.1.: Metaznaki w wyrażeniach regularnych.

Operator	Wyjaśnienie
^	Określa początek tekstu/linii
$	Określa koniec testu/linii
()	Grupowanie
|	Alternatywa (lub)
[]	Wymienia dozwolone znaki
[^]	Wymienia niedozwolone znaki
*	Poprzedni znak zostanie wybrany zero lub więcej razy
+	Poprzedni znak zostanie wybrany jeden lub więcej razy
?	Poprzedni znak zostanie wybrany zero lub jeden raz
{n}	Poprzedni znak zostanie wybrany n razy
.	Jakikolwiek znak oprócz nowej linii (\n)
\	Pozwala na użycie specjalnych znaków

Wymienione powyżej znaki (np. `^` czy .) mają specjalne znaczenie. W związku z tym, jeżeli chcemy wyszukać tekstu zawierającego specjalny znak, musimy użyć ukośnik wsteczny (`\`, ang. *backslash*). Istnieje wiele dodatkowych znaków specjalnych, np. `\n` - nowa linia, `\t` - tabulator, `\d` - każdy znak numeryczny (stałoprzecinkowy), `\s` - znak niedrukowalny, np. spacja, tabulator, nowa linia.

Sprawdźmy działanie wyrażeń regularnych na kilku przykładach. W pierwszym z nich określiliśmy nasz wzorzec jako `"^L"`, co oznacza, że interesują nas tylko elementy wektora `tekst3` rozpoczynające się od dużej litery `L`.

```
str_detect(tekst3, pattern = "^L")
#> [1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
```

Do określenia zakończenia wzorca służy metaznak `$`. Poniżej wyszukano elementy, które kończą się na `ena`.

```
str_detect(tekst3, pattern = "ena$")
#> [1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE
```

Operatory `()` i `|` można łączyć, aby zdefiniować alternatywy. Przykładowo, interesują nas elementy, które kończą się na `ena` lub `nna`.

```
str_detect(tekst3, pattern = "(ena|nna)$")
#> [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
```

W wyrażeniach regularnych można też stosować pewne skrótowe polecenia. W poniższym przypadku interesują nas elementy, które zawierają jakiekolwiek znaki od małego `a` do małego `z` oraz dużego `A` do dużego `Z`.

```
str_detect(tekst3, pattern = "[a-zA-Z]")
#> [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

Podobnie można określać wartości numeryczne - np. tylko elementy zawierające wartości od `0` do `9`.

```
str_detect(tekst3, pattern = "[0-9]")
#> [1] FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
```

Celem metaznaku `+` jest określenie, że poprzedni znak musi wystąpić jeden lub więcej razy. Poniżej interesują nas tylko takie elementy, w których litera `L` występuje raz lub więcej.

```
str_detect(tekst3, pattern = "L+")
#> [1] FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

W wyrażeniach regularnych metaznak `^` określa początek tekstu/linii, ale ma on też inne zastosowanie, gdy jest użyty w kwadratowym nawiasie. Przykładowo `[^L]` oznacza, że szukamy wszystkich elementów nie zawierających litery `L`. W poniższym przykładzie nie interesują nas elementy, które zaczynają się od jednej lub więcej litery `L`.

```
str_detect(tekst3, pattern = "^[^L]+")
#> [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
```

Metaznak . służy do określania jakikolwiek znaku, a ukośnik wsteczny (\) służy do określania innych znaków specjalnych. Dlatego też, jeżeli chcemy wyszukać elementów zawierających kropki (.) musimy połączyć ukośnik wsteczny z tym znakiem.

```
str_detect(tekst3, pattern = "\.")
#> Error: '\.' is an unrecognized escape in character string starting ""\."
```

Powyższy przykład daje jednak komunikat błędu - aby użyć ukośnik wsteczny do zasygnalizowania, że interesuje nas kropka musimy wprowadzić go dwa razy[4].

```
str_detect(tekst3, pattern = "\\.")
#> [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
```

Celem funkcji `str_detect()` jest wskazanie, który element spełnia dane zapytanie. Do wydzielenia elementu służy funkcja `str_subset()`.

```
str_subset(tekst3, pattern = "\\.")
#> [1] "1Lena.csv"
```

Umiejętności używania wyrażeń regularnych można trenować używając różnych zasobów internetowych, np. strony https://regexr.com/, https://regex101.com/, czy https://regexcrossword.com/. Pomocne w zrozumieniu bardziej zaawansowanych elementów wyrażeń regularncych może być też prezentacja Best of Fluent 2012: /Reg(exp){2}lained/: Demystifying Regular Expressions[5] oraz książka Mastering Regular Expressions (Friedl, 2006).

6.5. Wydzielanie tekstu - regex

Innym często spotykanym problemem w pracy z tekstem jest posiadanie długiego elementu tekstowego, z którego chcemy tylko wydobyć pewien

[4] https://xkcd.com/1638/
[5] https://www.youtube.com/watch?v=EkluES9Rvak

fragment. W sekcji 6.3 używaliśmy do tego pozycji, ale możemy zastosować również wzroce do tego celu.

```
tekst_pomiary = "Wrocław: 23.5, Bydgoszcz: 12.7, Toruń: 11.1, Lublin: 14.3"
```

Wektor `tekst_pomiary` zawiera tylko jeden element tekstowy, w którym wymienione są kolejne miasta i ich wartości. Wyobraźmy sobie, że interesują nas tylko nazwy miast zawarte w powyższym wektorze. Do wydzielania tekstu na podstawie wyrażeń regularnych służy funkcja `str_extract()`.

```
str_extract(tekst_pomiary, pattern = "[a-zA-Z]*")
#> [1] "Wroc"
```

Podaliśmy jako wzorzec wszyskie litery od małego `a` do małego `z` oraz dużego `A` do dużego `Z`. Niestety w efekcie otrzymaliśmy tylko Wroc - taka definicja wzorca obejmuje tylko litery z angielskiego alfabetu.

Aby to naprawić możemy dodać do tego wzorca polskie litery.

```
str_extract(tekst_pomiary, pattern = "[a-zA-ZąćęłńóśźżĄĆĘŁŃÓŚŹŻ]*")
#> [1] "Wrocław"
```

Tym razem otrzymaliśmy pełną nazwę pierwszego miasta, ale nie żadnego kolejnego. Funkcja `str_extract()` jest leniwa - po znalezieniu pierwszego pasującego fragmentu przestaje ona szukać dalej i przekazuje wynik.

Aby uzyskać wszystkie przypadki spełniające określony wzorzec należy użyć funkcji `str_extract_all()`.

```
str_extract_all(tekst_pomiary, pattern = "[a-zA-ZąćęłńóśźżĄĆĘŁŃÓŚŹŻ]*")
#> [[1]]
#>  [1] "Wrocław"   ""          ""          ""
#>  [5] ""          ""          ""          ""
#>  [9] ""          "Bydgoszcz" ""          ""
#> [13] ""          ""          ""          ""
#> [17] ""          ""          "Toruń"     ""
#> [21] ""          ""          ""          ""
#> [25] ""          ""          ""          "Lublin"
#> [29] ""          ""          ""          ""
#> [33] ""          ""          ""
```

Efektem jej działania są wszystkie nazwy miast z wektora `tekst_pomiary`, ale też wiele elementów pustych. Dlaczego? W powyższym wzorcu użyliśmy metaznaku `*`, który szuka wystąpienia zdefiniowanych znaków zero lub więcej razy. Gdy napotkany jest zdefiniowany znak sprawdzane jest jego kolejne wystąpienie, aż do momentu, gdy pojawi się jakiś inny znak. W efekcie zwrócony został np. `"Wrocław"`. Metaznak `*` w przypadku, gdy zdefiniowanego znaku nie ma (wystąpił zero razy) zwraca pusty element.

Jeżeli interesują nas tylko fragmenty wektora zawierające tekst musimy użyć metaznaku `+`.

```
str_extract_all(tekst_pomiary, pattern = "[a-zA-ZąćęłńóśźżĄĆĘŁŃÓŚŹŻ]+")
#> [[1]]
#> [1] "Wrocław"   "Bydgoszcz" "Toruń"     "Lublin"
```

Wyobraźmy sobie, że otrzymaliśmy rozszerzoną wersję poprzednich danych, która tym razem zawiera dwa dodatkowe miasta - Gorzów Wielkopolski i Zieloną Górę.

```
tekst_pomiary2 = "Wrocław: 23.5, Bydgoszcz: 12.7, Toruń: 11.1, Lublin: 14.3,
Gorzów Wielkopolski: 20, Zielona Góra: 19"
```

Nadal interesuje nas wydzielenie nazw miast, więc próbujemy użyć kodu, który stworzyliśmy powyżej.

```
str_extract_all(tekst_pomiary2, pattern = "[a-zA-ZąćęłńóśźżĄĆĘŁŃÓŚŹŻ]+")
#> [[1]]
#> [1] "Wrocław"      "Bydgoszcz"    "Toruń"
#> [4] "Lublin"       "Gorzów"       "Wielkopolski"
#> [7] "Zielona"      "Góra"
```

Niestety w efekcie otrzymujemy osiem elementów, gdzie `"Gorzów"` jest innym elementem niż `"Wielkopolski"`. Zdefiniowany przez nas wzorzec nie brał pod uwagę możliwości wystąpienia spacji. Możemy naprawić tę sytuację w poniższy sposób.

```
str_extract_all(tekst_pomiary2,
       pattern = "[a-zA-ZąćęłńóśźżĄĆĘŁŃÓŚŹŻ]+[\\s]?[a-zA-ZąćęłńóśźżĄĆĘŁŃÓŚŹŻ]*")
#> [[1]]
#> [1] "Wrocław"             "Bydgoszcz"
#> [3] "Toruń"               "Lublin"
#> [5] "Gorzów Wielkopolski" "Zielona Góra"
```

Teraz szukamy wystąpienia liter co najmniej raz lub więcej (`+`), następnie wystąpienia spacji zero razy lub raz (`[\\s]?`) i kończymy na sprawdzeniu wystąpienia tekstu zero razy lub więcej (`*`).

Podobnie jak w każdym powyższym przypadku, efekt działania funkcji może być użyty do stworzenia nowego obiektu.

```
miasta_pomiary2 = str_extract_all(tekst_pomiary2,
       pattern = "[a-zA-ZĄĆĘŁŃÓŚŹŻąćęłńóśźż]+[\\s]?[a-zA-ZĄĆĘŁŃÓŚŹŻąćęłńóśźż]*")
miasta_pomiary2
#> [[1]]
#> [1] "Wrocław"             "Bydgoszcz"
#> [3] "Toruń"               "Lublin"
#> [5] "Gorzów Wielkopolski" "Zielona Góra"
```

6.6. Zamiana tekstu - regex

Innym przykładem działania na tekście jest zamiana wybranych jego elementów.

```
tekst_pomiary3 = "Wrocław: 23.5, Bydgoszcz: 12.7, Toruń: 11.1, Lublin: 14.3"
```

Powyższy obiekt `tekst_pomiary3` zawiera nazwy miast i wartości pomiarów przedstawione zgodnie z amerykańskim standardem, gdzie kropka oddziela wartości dziesiętne, a przecinek kolejne elementy. Aby zamienić wybrany wzorzec w tekście (np. kropkę na przecinek) możemy użyć funkcji `str_replace()`, w której podajemy obiekt tekstowy, szukany wzorzec oraz jego zamianę.

```
str_replace(tekst_pomiary3,
            pattern = ".",
            replacement = ",")
#> [1] ",rocław: 23.5, Bydgoszcz: 12.7, Toruń: 11.1, Lublin: 14.3"
```

Efekt działania tego kodu nie jest jednak zgodny z naszymi oczekiwaniami. Zamiast zamiany wszyskich kropek na przecinki, nastąpiła zamiana pierwszego znaku w tekście (litery `W`) na przecinek. Wynika to ze znaczenia metaznaku `.`, który reprezentuje jakikolwiek znak oprócz nowej linii. Żeby naprawić tę sytuację musimy użyć ukośnika wstecznego.

```
str_replace(tekst_pomiary3,
            pattern = "\\.",
            replacement = "\\,")
#> [1] "Wrocław: 23,5, Bydgoszcz: 12.7, Toruń: 11.1, Lublin: 14.3"
```

Funkcja `str_replace()`, podobnie jak `str_extract()`, jest leniwa i zamienia tylko pierwsze wystąpienie wzorca. Do zamiany wszystkich przypadków trzeba użyć funkcji `str_replace_all()`.

```
str_replace_all(tekst_pomiary3,
                pattern = "\\.",
                replacement = "\\,")
#> [1] "Wrocław: 23,5, Bydgoszcz: 12,7, Toruń: 11,1, Lublin: 14,3"
```

W przypadku, gdy interesuje nas zarówno zamiana kropek na przecinki oraz przecinków na średniki, musimy zacząć od tej drugiej zamiany.

```
tekst_pomiary4 = str_replace_all(tekst_pomiary3,
                                 pattern = "\\,",
                                 replacement = "\\;")
```

Nowy obiekt `tekst_pomiary4` oddziela kolejne miasta średnikami. Teraz na jego podstawie możliwa jest zamiana kropek na przecinki w sposób opisany powyżej.

```
str_replace_all(tekst_pomiary4,
                pattern = "\\.",
                replacement = "\\,")
#> [1] "Wrocław: 23,5; Bydgoszcz: 12,7; Toruń: 11,1; Lublin: 14,3"
```

Funkcje jakie jak `str_replace()` czy `str_replace_all()` mogą być też stosowane do usuwania fragmentów tekstu. Do tego celu można zdefiniować wzorzec jaki chcemy usunąć, a jako jego zamianę tekst pusty (`""`).

```
str_replace_all(tekst_pomiary4,
                pattern = "[a-zA-ZąćęłńóśźżĄĆĘŁŃÓŚŹŻ]+",
                replacement = "")
#> [1] ": 23.5; : 12.7; : 11.1; : 14.3"
```

6.7. Wyszukiwanie plików

Umiejętności związane z obsługą wyrażeń regularnych przydają się też w przypadku wyszukiwania plików zawierających określony tekst w nazwie lub specyficzne rozszerzenie. Jest to szczególnie przydatne, gdy posiadamy wiele plików na komputerze, które chcemy następnie przetwarzać w sposób automatyczny.

Do wyświetlania nazw plików znajdujących się w wybranym folderze służy funkcja `dir()`. Przykładowo poniższa linia kodu wyświetla wszyskie pliki znajdujące się w folderze `"pliki"`[6].

```
dir("pliki")
#>  [1] "dane_meteo.csv"  "dane_meteo.rds"
#>  [3] "dane_meteo.xlsx" "dane_meteo2.csv"
#>  [5] "dokument.docx"   "kod.R"
#>  [7] "list.txt"        "mapa.png"
#>  [9] "obrazek.png"     "zdjecie.jpg"
```

W przypadku, gdy interesują nas tylko pliki o wybranym rozszerzeniu możemy użyć argumentu `pattern` i zdefiniować wzorzec.

```
dir("pliki", pattern = "*\\.png$")
#> [1] "mapa.png"    "obrazek.png"
```

W powyższym przykładzie zostaną wybrane tylko pliki o jakiejkolwiek nazwie, ale kończące się na rozszerzenie `.png`. Metaznak `$` użyty w tym przypadku zapogiega sytuacji, gdy tekst `.png` znajduje się w środku nazwy pliku.

Do znalezienia plików o kilku rozszerzeniach można użyć metaznaków `()` i `|`.

```
dir("pliki", pattern = "*\\.(png|jpg)$")
#> [1] "mapa.png"    "obrazek.png" "zdjecie.jpg"
```

Domyślnie funkcja `dir()` pokazuje zawartość wybranego folderu, aby jednak poznać jego pełną ścieżkę względną należy określić argument `full.names` na `TRUE`.

[6] Folder o tej nazwie znajduje się w folderze roboczym.

```r
dir("pliki", pattern = "*\\.(png|jpg)$", full.names = TRUE)
#> [1] "pliki/mapa.png"    "pliki/obrazek.png"
#> [3] "pliki/zdjecie.jpg"
```

6.8. Zadania

1) Plik tekstowy zawiera listę pomiarów, w której piąty i czwarty znak od końca oznacza symbol chemiczny pomierzonego pierwiastka, przykładowo:

```
"TERYT 18; podkarpackie; Rzeszów; 0.2 He; A"
"TERYT 22; pomorskie;   Gdańsk; 12 C ; B"
```

Napisz kod, który będzie wydzielał symbole chemiczne pomierzonych pierwiastków.

2) Napisz funkcję nazywającą się `horoskop`, która przyjmuje dwa argumenty `imie` (pierwsze imię, tekst) oraz `miesiac` (miesiąc urodzin, liczba). Funkcja ma zwrócić tekst "Osoba o imieniu 'imie' będzie miała jutro szczęście." w przypadku, gdy argument `miesiac` jest liczbą parzystą oraz "Osoba o imieniu 'imie' będzie miała jutro nieszczęście." jeżeli argument `miesiac` jest liczbą nieparzystą.
3) Rozbuduj funkcję `horoskop` poprzez sprawdzenie pierwszej litery podanego imienia. Jeżeli pierwsza litera imienia to `K`, `M`, lub `Z` wówczas wyświetli się zawsze tekst "Osoba o imieniu 'imie' będzie miała jutro szczęście.", bez względu na podany miesiąc.
4) Efektem zbierania pomiarów temperatury okazał być się plik tekstowy, który zawiera datę pomiaru oraz wartość. W jaki sposób możliwe jest wydzielenie tylko dat w takiej sytuacji? Poniżej znajduje się fragment przykładowych danych wejściowych.

```
"2019-03-11: 23.5, 19/03/12: 12.7, 2019.03.13: 11.1, 2019-marzec-14: 14.3"
```

5) Co należałoby zrobić, aby wydzielić tylko wartości pomiarów w powyższym przypadku? Stwórz nowy obiekt `wartosci` zawierający te pomiary. Jakiej klasy powinien być wyjściowy obiekt?
6) Posiadasz wektor `wsp` zawierający współrzedne geograficzne szeregu miast w formacie DMS (Stopnie, Minuty, Sekundy). Wydziel tylko wartości stopnii z tej reprezentacji. Poniżej znajduje się fragment przykładowych danych wejściowych.

```
wsp = c("52°24'N 16°55'E", "53°08'07"N 23°08'44"E", "39°6'N 84°31'W")
```

7) Stwórz funkcję, która przyjmując przykładowe dane z poprzedniego zadania zamieni współrzędne na format w postaci stopni dziesiętnych (np. `52°24'N` w formacie DMS to `52.4` w stopniach dziesiętnych).

7. Złożone obiekty

W rozdziale 5 omówiono wektory atomowe, które są obiektami jednowymiarowymi zawierającymi tylko jeden typ danych. Ten rozdział jest poświęcony pozostałymi trzema podstawowymi klasami obiektów w R - macierzami, ramkami danych i listami (sekcje 7.1, 7.2, i 7.3).

7.1. Macierze

Macierze (ang. *matrix*), podobnie jak wektory, są obiektami homogenicznymi - jedna macierz może przyjmować dane tylko jednego typu. Od wektorów różnią się jednak tym, że są dwuwymiarowe - wartości ułożone są w kolejnych wierszach i kolumnach. Macierze są używane do różnorodnych obliczeń matematycznych i statystycznych. W uproszczeniu można o nich myśleć jako o reprezentacji komputerowej zdjęcia lub mapy.

W R istnieją też wielowiarowe obiekty podobne do macierzy zwane matrycami (ang. *array*).

7.1.1. Tworzenie

Tworzenie macierzy odbywa się poprzez użycie funkcji `matrix()`, która przyjmuje wartości wektora jako pierwszy argument, a następnie informacje o wymiarach w postaci liczby wierszy (`nrow`) i liczby kolumn (`ncol`).

```
macierz1 = matrix(1:12, nrow = 4, ncol = 3)
macierz1
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
```

Domyślnie w R wartości wpisywane są do macierzy do kolejnych kolumn startując od lewej strony. Możliwe jest jednak ustawienie argumentu `byrow = TRUE` co powoduje wpisywanie podanych wartości dla kolejnych wierszy zamiast kolejnych kolumn.

```
macierz2 = matrix(1:12, nrow = 4, ncol = 3, byrow = TRUE)
macierz2
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
#> [3,]    7    8    9
#> [4,]   10   11   12
```

7.1.2. Podstawowe funkcje

Sprawdzenie jakiej klasy jest podany obiekt odbywa się używając funkcji `class()`.

```
class(macierz2)
#> [1] "matrix"
```

Więcej informacji na temat obiektu można poznać używając funkcji `str()`. Jej działanie na macierzy jest bardzo podobne do wyniku na wektorach (zobacz sekcję 5.3) - wyświetlony zostaje typ obiektu (np. `int`), jego wymiary (np. `[1:4, 1:3]`) i kilka przykładowych wartości (np. `1 4 7 10 2 5 8 11 3 6`).

```
str(macierz2)
#>  int [1:4, 1:3] 1 4 7 10 2 5 8 11 3 6 ...
```

Macierz może przyjmować tylko jeden typ obiektów, co można sprawdzić używając funkcji `typeof()`.

```
typeof(macierz2)
#> [1] "integer"
```

Wektory atomowe mają tylko jeden wymiar więc ich długość oznacza liczbę elementów i może być sprawdzona używając funkcji `length()`. W przypadku macierzy możliwe jest dodatowo sprawdzenie liczby występujących wierszy (`nrow`) i kolumn (`ncol`).

```
nrow(macierz2)
#> [1] 4
ncol(macierz2)
#> [1] 3
```

Domyślnie macierze nie zawierają nazw kolumn ani wierszy.

```
colnames(macierz2)
#> NULL
```

Sprawdzenie czy dodanie nazw kolumn jest jednak możliwe używając funkcji `colnames()`. Może to pozwolić w przyszłości na wydzielanie konkretnych wartości na podstawie nazw kolumn.

```
colnames(macierz2) = c("a", "b", "c")
macierz2
#>       a  b  c
#> [1,]  1  2  3
#> [2,]  4  5  6
#> [3,]  7  8  9
#> [4,] 10 11 12
colnames(macierz2)
#> [1] "a" "b" "c"
```

7.1.3. Wydzielanie

Podobnie jak w przypadku wektorów (rozdział 5), macierze można wydzielać używając operatora `[]`. W tym wypadku odbywa się to jednak w oparciu o dwa indeksy - jeden dla wiersza, drugi dla kolumny - `[wiersz, kolumna]`.

Przykładowo, poniżej zostaną wybrane tylko wartości znajdujące się w pierwszy i drugim wierszy oraz pierwszej i trzeciej kolumnie.

```
macierz2[c(1, 2), c(1, 3)]
#>      a c
#> [1,] 1 3
#> [2,] 4 6
```

Do wydzielania macierzy czy ramek danych też często przydatne jest używanie elementu pustego. Pozwala on na wybór wszystkich wartości w danym wymiarze. Na poniższy przykładzie zostały wybrane wiersze jeden i dwa oraz, z uwagi na element pusty, wszystkie kolumny.

```
macierz2[c(1, 2), ]
#>      a b c
#> [1,] 1 2 3
#> [2,] 4 5 6
```

Element pusty można też zastosować do wybrania wszystkich wierszy.

```
macierz2[, c(1, 3)]
#>       a  c
#> [1,]  1  3
#> [2,]  4  6
#> [3,]  7  9
#> [4,] 10 12
```

Wszystkie pozostałe sposoby wydzielania opisane dla wektorów w sekcji 5.6 działają również na macierzach. Możeliwe jest więc używanie wektora logicznego czy nazw kolumn.

```
macierz2[, c(TRUE, FALSE, TRUE)]
#>       a  c
#> [1,]  1  3
#> [2,]  4  6
#> [3,]  7  9
#> [4,] 10 12
macierz2[, c("a", "c")]
#>       a  c
#> [1,]  1  3
#> [2,]  4  6
#> [3,]  7  9
#> [4,] 10 12
```

7.1.4. Łączenie

Łączenie wektorów odbywa się używając jednej funkcji `c()`. W przypadku macierzy występują jednak dwa wymiary - możliwe jest połączenie macierzy wierszami lub kolumnami. W efekcie istnieją do tego dwie oddzielne funkcje `rbind()` i `cbind()`.

Pierwsza z nich łączy macierze wierszami.

```
macierz3 = rbind(macierz1, macierz2)
macierz3
#>       a  b  c
#> [1,]  1  5  9
#> [2,]  2  6 10
#> [3,]  3  7 11
#> [4,]  4  8 12
#> [5,]  1  2  3
#> [6,]  4  5  6
#> [7,]  7  8  9
#> [8,] 10 11 12
```

Druga dokleja obiekty kolumnami.

```
macierz4 = cbind(macierz1, macierz2)
macierz4
#>              a  b  c
#> [1,] 1 5  9  1  2  3
#> [2,] 2 6 10  4  5  6
#> [3,] 3 7 11  7  8  9
#> [4,] 4 8 12 10 11 12
```

W przypadku, gdy chcemy połączyć kilka wektorów różnych typów działają dokładnie takie same reguły jak w przypadku wektorów atomowych (sekcja 5.9).

7.2. Ramki danych

Ramki danych (ang. *data frame*) mają dużo podobieństw z macierzami. Są to obiekty dwuwymiarowe, składające się z kolumn i wierszy. Główną różnicą pomiędzy macierzą a ramką danych jest to, że pierwsza z nich przyjmuje tylko dane jednego typu, podczas gdy druga może się składać z danych róznych typów.

Ramka danych jest zbudowana z kolumn (wektorów) o równej długości. Ten typ obiektu jest głównie wykorzystywany do różnorodnej analizy danych. Ramki danych przypominają w swojej strukturze arkusze kalkulacyjne czy bazy danych.

7.2.1. Tworzenie

Stworzenie nowej ramki danych możliwe jest używając funkcji `data.frame()`, w której podawane są nazwy kolejnych kolumn (np. `wek_log`) oraz ich wartości (np. `c(TRUE, FALSE, FALSE)`).

```
ramka1 = data.frame(wek_log = c(TRUE, FALSE, FALSE),
                    wek_cal = c(5L, -7L, 12L),
                    wek_zmi = c(5.3, -7.1, 1.1),
                    wek_zna = c("kot", "pies", "nosorożec"),
                    stringsAsFactors = FALSE)
ramka1
#>   wek_log wek_cal wek_zmi   wek_zna
#> 1    TRUE       5     5.3       kot
#> 2   FALSE      -7    -7.1      pies
#> 3   FALSE      12     1.1 nosorożec
```

W powyższym przykładzie, `ramka1` składa się z czterech kolumn o długości trzy. Każda z tych kolumn ma inny typ - logiczny, liczby całkowitej, liczby zmiennoprzecinkowej oraz znakowy. Domyślnie funkcja `data.frame` wykonuje jeszcze jedną operację w tle - zamienia ona wszystkie dane o typie znakowym na typ czynnikowy (sekcja 5.11). W większości przypadków nie jest to porządane działanie - dlatego też warto wyłączyć tę konwersję używając argumentu `stringsAsFactors = FALSE`.

Obiekty klasy ramka danych są też zazwyczaj wynikiem wczytywania zewnętrznych plików do R, np. w formacie `.csv` czy `.xlsx`. Więcej informacji na ten temat można znaleźć w rozdziale 9.

7.2.2. Podstawowe funkcje

Oficjalnie klasa ramki danych jest określana jako `data.frame`.

```
class(ramka1)
#> [1] "data.frame"
```

Sprawdzenie struktury ramki danych pozwala na szybkie poznanie kilku róznych cech wejściowego obiektu. Pierwszą informacją jest klasa obiektu (`data.frame`), liczba wierszy (`3 obs.` - trzy obserwacje) i liczba kolumn (`4 variables` - cztery zmienne). Następnie, dla kolejnych kolumn są określane ich nazwy, typy danych oraz przykładowe wartości.

```
str(ramka1)
#> 'data.frame':    3 obs. of  4 variables:
#>  $ wek_log: logi  TRUE FALSE FALSE
#>  $ wek_cal: int  5 -7 12
#>  $ wek_zmi: num  5.3 -7.1 1.1
#>  $ wek_zna: chr  "kot" "pies" "nosorożec"
```

Podobnie jak w przypadku macierzy, ramki danych mają dwa wymiary, których długość można sprawdzić używając funkcji `nrow` i `ncol`.

```
nrow(ramka1)
#> [1] 3
ncol(ramka1)
#> [1] 4
```

W przeciwieństwie jednak do macierzy, ramki danych zawsze posiadają nazwy kolumn.

```
colnames(ramka1)
#> [1] "wek_log" "wek_cal" "wek_zmi" "wek_zna"
```

Ich zmiana również jest możliwa używając funkcji `colnames()`.

```
colnames(ramka1) = c("log", "cal", "zmi", "zna")
ramka1
#>     log cal  zmi       zna
#> 1  TRUE   5  5.3       kot
#> 2 FALSE  -7 -7.1      pies
#> 3 FALSE  12  1.1 nosorożec
colnames(ramka1)
#> [1] "log" "cal" "zmi" "zna"
```

7.2.3. Wydzielanie

Do wydzielania elementów z ramki danych może służyć kilka narzędzi, między innymi, operator `$`, operator `[]` oraz funkcja `subset()`.

Operator `$` pozwala na wybranie zmiennej (kolumny) na podstawie jej nazwy.

```
ramka1$zmi
#> [1]  5.3 -7.1  1.1
ramka1$zna
#> [1] "kot"       "pies"      "nosorożec"
```

W efekcie otrzymywany jest jednak inna klasa - w powyższych przykładzach są to wektory.

W przypadku ramek danych operator `[]` wymaga podania dwóch argumentów - jednego dla wierszy (obserwacji) oraz jednego dla kolumn (zmiennych) - `[wiersze, kolumny]`.

```
ramka1[c(1, 3), c(1, 2)]
#>     log cal
#> 1  TRUE   5
#> 3 FALSE  12
```

Do wydzielania można też wykorzystać operatory logiczne: `==`, `%in%`, `!=`, `>`, `>=`, `<`, `<=`, `&`, `|`.

```
ramka1[ramka1$zmi > 0, c(1, 3)]
#>     log zmi
#> 1  TRUE 5.3
#> 3 FALSE 1.1
```

Powyżej wybrano tylko pierwszą i trzecią kolumnę oraz wiersze, dla których kolumna `zmi` miała wartość wyższą niż 0.

Zapytania też można łączyć, np. wybierając tylko te wiersze gdzie wartość `cal` jest wyższa niż 6 lub niższa niż -6.

```
ramka1[ramka1$cal > 6 | ramka1$cal < -6, ]
#>     log cal  zmi       zna
#> 2 FALSE  -7 -7.1      pies
#> 3 FALSE  12  1.1 nosorożec
```

Poniżej wybrano natomiast wszystkie kolumny, ale tylko wiersz gdzie zmienna `zna` przyjęła wartość `"kot"`.

```
ramka1[ramka1$zna == "kot", ]
#>    log cal zmi zna
#> 1 TRUE   5 5.3 kot
```

Aby wybrać więcej niż jedną zmienną należy użyć funkcji `%in%`.

```
ramka1[ramka1$zna %in% c("kot", "pies"), ]
#>     log cal  zmi  zna
#> 1  TRUE   5  5.3  kot
#> 2 FALSE  -7 -7.1 pies
```

Alternatywą do wydzielania ramek danych na podstawie zapytania logicznego jest użycie funkcji `subset()`. Używając tej funkcji powyższe zapytanie można przestawić jako:

```
subset(ramka1, zna %in% c("kot", "pies"))
#>     log cal  zmi  zna
#> 1  TRUE   5  5.3  kot
#> 2 FALSE  -7 -7.1 pies
```

Podobnie jak w przypadku wektorów (sekcje 5.7 i 5.8), wydzielanie ramek danych może służyć do wyświetlenia wybranych wartości, ale też ich wydzielenia i przypisania oraz modyfikowania.

7.2.4. Łączenie

Łączenie ramek danych przypomina łączenie macierzy używając funkcji `rbind()` i `cbind()`. Jednocześnie należy pamiętać, że łączenie wierszy (`rbind()`) wymaga posiadania kolumn o tych samych nazwach w obu obiektach.

```
ramka2 = data.frame(log = TRUE, cal = 2L, zmi = 2.3, zna = "żółw")
ramka2
#>    log cal zmi  zna
#> 1 TRUE   2 2.3 żółw

rbind(ramka1, ramka2)
#>     log cal  zmi       zna
```

```
#> 1  TRUE   5  5.3       kot
#> 2 FALSE  -7 -7.1      pies
#> 3 FALSE  12  1.1 nosorożec
#> 4  TRUE   2  2.3      żółw
```

Ograniczeniem łącznia kolumn jest posiadanie tej samej długości każdej kolumny.

```
ramka3 = data.frame(zmi2 = c(4.3, 2.6, 7.4))
ramka3
#>   zmi2
#> 1  4.3
#> 2  2.6
#> 3  7.4

cbind(ramka1, ramka3)
#>     log cal  zmi       zna zmi2
#> 1  TRUE   5  5.3       kot  4.3
#> 2 FALSE  -7 -7.1      pies  2.6
#> 3 FALSE  12  1.1 nosorożec  7.4
```

Funkcje `rbind()` i `cbind()` łączą obiekty nie zmieniając kolejności występujących w nich wartości. Bardziej zaawansowanymi sposobami łączenia ramek danych są różnorodne operacje łączenia (ang. *joins*), np. `left_join()` czy `inner_join()` z pakietu **dplyr**. Więcej na ten temat można znaleźć w rozdziale "Relational data"[1] książki Advanced R (Wickham, 2014).

7.3. Listy

Ostatnią podstawową klasą obiektów w R są listy (ang. *list*). Ta klasa pozwala na przechowywanie obiektów o różnych typach i różnej długości.

7.3.1. Tworzenie

Do tworzenia list służy funkcja `list()`, która przyjmuje jako argumenty kolejne obiekty, które mają się w niej znaleźć.

```
lista1 = list(c(TRUE, FALSE),
              c(5L, -7L),
              c(5.3),
              c("kot", "pies", "nosorożec"))
lista1
#> [[1]]
#> [1]  TRUE FALSE
#>
#> [[2]]
#> [1]  5 -7
#>
#> [[3]]
#> [1] 5.3
#>
#> [[4]]
#> [1] "kot"       "pies"      "nosorożec"
```

Przykładowa powyższa lista składa się z czterech elementów, o długości jeden, dwa lub trzy, i o różnorodnych typach danych.

Inną ważną właściwością list jest możliwość ich zagnieżdzania - jedna lista może przechowywać kolejną, która jest w stanie przechowywać następną... Z tego powodu listy są czasami nazywane wektorami rekurencyjnymi.

```
zlozona_lista1 = list(list(list(lista1)))
str(zlozona_lista1)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ :List of 4
#>   .. .. ..$ : logi [1:2] TRUE FALSE
#>   .. .. ..$ : int [1:2] 5 -7
#>   .. .. ..$ : num 5.3
#>   .. .. ..$ : chr [1:3] "kot" "pies" "nosorożec"
```

7.3.2. Podstawowe funkcje

Listy są przedstawane jako klasa `list`.

```
class(lista1)
#> [1] "list"
```

W ich wypadku funkcja `str()` wyświetla klasę (`List`), liczbę elementów w liście (4) oraz wypisuje kolejne elementy listy, ich typy, wymiary i przykładowe wartości.

```
str(lista1)
#> List of 4
#>  $ : logi [1:2] TRUE FALSE
#>  $ : int [1:2] 5 -7
#>  $ : num 5.3
#>  $ : chr [1:3] "kot" "pies" "nosorożec"
```

Listy nie zawierają wierszy czy kolumn, dlatego do sprawdzenia liczby elementów w liście służy tylko funkcja `length()`.

```
length(lista1)
#> [1] 4
```

Kolejne elementy znajdujące się w liście mogą przyjmować wbrane nazwy. Można je sprawdzić czy zmienić używając funkcji `names()`.

```
names(lista1) = c("log", "cal", "zmi", "zna")
lista1
#> $log
#> [1]  TRUE FALSE
#>
#> $cal
#> [1]  5 -7
#>
#> $zmi
#> [1] 5.3
#>
#> $zna
#> [1] "kot"       "pies"      "nosorożec"
names(lista1)
#> [1] "log" "cal" "zmi" "zna"
```

7.3.3. Łączenie

Łączenie list może odbywać się na dwa postawowe sposoby. W pierwszym, używając funkcji `c()` następuje dołączenie elementów jednej listy do drugiej.

```
lista2 = c(lista1, lista1)
str(lista2)
#> List of 8
#>  $ log: logi [1:2] TRUE FALSE
#>  $ cal: int [1:2] 5 -7
#>  $ zmi: num 5.3
#>  $ zna: chr [1:3] "kot" "pies" "nosorożec"
#>  $ log: logi [1:2] TRUE FALSE
#>  $ cal: int [1:2] 5 -7
#>  $ zmi: num 5.3
#>  $ zna: chr [1:3] "kot" "pies" "nosorożec"
```

Efektem nadal jest jedna lista, ale składająca się z większej liczby elementów.

Drugim sposobem jest użycie funkcji `list()`. W tym przypadku tworzona jest nowa, nadrzędna lista, która zawiera dwie wcześniejsze listy.

```
lista3 = list(lista1, lista1)
str(lista3)
#> List of 2
#>  $ :List of 4
#>   ..$ log: logi [1:2] TRUE FALSE
#>   ..$ cal: int [1:2] 5 -7
#>   ..$ zmi: num 5.3
#>   ..$ zna: chr [1:3] "kot" "pies" "nosorożec"
#>  $ :List of 4
#>   ..$ log: logi [1:2] TRUE FALSE
#>   ..$ cal: int [1:2] 5 -7
#>   ..$ zmi: num 5.3
#>   ..$ zna: chr [1:3] "kot" "pies" "nosorożec"
```

7.3.4. Wydzielanie

Wydzielanie list może mieć miejsce używając jednego z trzech operatorów - `[]`, `[[]]`, oraz `$`.

Operator `[]` wydziela wybrane elementy z listy, ale jednocześnie dalej zwraca w wyniku obiekt klasy lista. Wyobraź sobie, że masz torbę zawierający cztery przedmioty (*listę zawierającą cztery elementy*) i chcesz zostawić w plecaku tylko pierwszy i drugi z nich.

```
lista4 = lista1[c(1, 2)]
lista4
#> $log
#> [1]  TRUE FALSE
#>
#> $cal
#> [1]  5 -7
str(lista4)
#> List of 2
#>  $ log: logi [1:2] TRUE FALSE
#>  $ cal: int [1:2] 5 -7
```

W efekcie wynikowy obiekt nadal jest listą, ale z mniejszą liczbą elementów.

Do wydobycia wartości z listy służą operatory `[[]]` oraz `$`. Pierwszy z nich wydobywa wartości na podstawie ich położenia i w efekcie otrzymywany jest obiekt znajdujący się wewnątrz listy. W poniższym przykładzie, wydzielany jest czwarty element z obiektu `lista1`.

```
lista5 = lista1[[4]]
lista5
#> [1] "kot"       "pies"      "nosorożec"
str(lista5)
#>  chr [1:3] "kot" "pies" "nosorożec"
```

Czwarty element w `lista1` jest wektorem znakowym o długości trzy. W przypadku, gdy wybrany element listy jest innej klasy to jest on również zwracany. Poniżej drugi element `listy3` jest również listą - wyobraź to sobie jako wyciągnięcie jednej torby, która znajduje się wewnątrz innej.

```
lista6 = lista3[[2]]
lista6
#> $log
#> [1]  TRUE FALSE
#>
#> $cal
#> [1]  5 -7
#>
#> $zmi
#> [1] 5.3
#>
```

```
#> $zna
#> [1] "kot"       "pies"      "nosorożec"
str(lista6)
#> List of 4
#>  $ log: logi [1:2] TRUE FALSE
#>  $ cal: int [1:2] 5 -7
#>  $ zmi: num 5.3
#>  $ zna: chr [1:3] "kot" "pies" "nosorożec"
```

Ostatni operator, `$`, wydziela wartości na podstawie ich nazw.

```
lista1$zna
#> [1] "kot"       "pies"      "nosorożec"
```

7.4. Zmiany klas

R ma też szereg pomocnicznych służących do zmian istniejących klas. Te funcje rozpoczynają się od `as.` a następnie zawierają nazwę klasy do której chcemy przetworzyć wejściowy obiekt, np. `as.vector()`, `as.matrix()`, `as.data.frame()`, `as.list()`.

Działanie tych funkcji jednak jest bardzo różne w zależności o klasy wejściowego obiektu. Zobaczmy to na przykładzie obiektu `macierz1`.

```
macierz1
#>      [,1] [,2] [,3]
#> [1,]    1    5    9
#> [2,]    2    6   10
#> [3,]    3    7   11
#> [4,]    4    8   12
class(macierz1)
#> [1] "matrix"
```

Jego zamiana na ramkę danych odbywa się używając funkcji `as.data.frame()`. W efekcie nowy obiekt ma takie same wymiary (cztery wiersze i trzy kolumny) oraz te same wartości. Zauważalną zmianą jest jednak automatyczne dodane nazw wierszy (`1`, `2`, `3`, `4`) i nazw kolumn (`V1`, `V2`, `V3`).

```
ramka_z_m1 = as.data.frame(macierz1)
ramka_z_m1
#>   V1 V2 V3
#> 1  1  5  9
#> 2  2  6 10
#> 3  3  7 11
#> 4  4  8 12
class(ramka_z_m1)
#> [1] "data.frame"
```

Zupełnie inny efekt będzie miało natomiast zamienienie macierzy i ramki danych na listy.

```
lista_z_m1 = as.list(macierz1)
lista_z_r1 = as.list(ramka_z_m1)
```

W pierwszym przypadku powstanie lista zawierająca dwanaście elementów.

```
str(lista_z_m1)
#> List of 12
#>  $ : int 1
#>  $ : int 2
#>  $ : int 3
#>  $ : int 4
#>  $ : int 5
#>  $ : int 6
#>  $ : int 7
#>  $ : int 8
#>  $ : int 9
#>  $ : int 10
#>  $ : int 11
#>  $ : int 12
```

W drugim przypadku efektem będzie lista składająca się z trzech wektorów, które reprezentują kolejne kolumny z poprzedniej ramki danych.

```
str(lista_z_r1)
#> List of 3
#>  $ V1: int [1:4] 1 2 3 4
```

```
#>  $ V2: int [1:4] 5 6 7 8
#>  $ V3: int [1:4] 9 10 11 12
```

7.5. Inne klasy obiektów

W tym oraz 5 rozdziale zostały wymienione i opisane cztery podstawowe klasy obiektów w R - wektory atomowe, macierze, natomiast listy i ramki danych. R zawiera jednak znacznie więcej klas obiektów, a co więcej - każda osoba może stworzyć swoją własną klasę obiektów.

Poniżej zostały stworzone trzy nowe wektory - znakowy (`wek_tkt`), numeryczny (`wek_num`) i dat (`wek_dat`).

```
wek_tkt = c("kot", "pies", "nosorożec")
wek_num = c(4, 6, 8)
wek_dat = as.Date(c("2019-04-10", "2019-04-12", "2019-04-14"))
```

Czy można na nich wyliczyć średnią w ten sam sposób? Raczej nie - wyliczenie średniej z tekstu nie jest jednoznacznie możliwe, wyliczenie średniej z wartości numerycznych powinno dać wartość numeryczną, a wyliczenie średniej z dat - również datę.

```
mean(wek_tkt)
#> Warning in mean.default(wek_tkt): argument is not
#> numeric or logical: returning NA
#> [1] NA
mean(wek_num)
#> [1] 6
mean(wek_dat)
#> [1] "2019-04-12"
```

Powyższe wyniki są poprawne, ale każdy z nich zwraca inny rodzaj wyniku. Jest to możliwe dzięki tzw. metodom (ang. *methods*). Metoda to sposób w jaki zachowuje się funkcja w zależności od tego jakiej klasy będzie obiekt wejściowy.

```
methods(mean)
#> [1] mean.Date     mean.default  mean.difftime
#> [4] mean.POSIXct  mean.POSIXlt
#> see '?methods' for accessing help and source code
```

Przykładowo, funkcja `mean()` ma pięć metod:

- `.Date` - obsługującą daty
- `.difftime` - obsługującą czas trwania
- `.POSIXct` oraz `.POSIXlt` - obsługujące czas
- `.default` - domyślna metoda

W momencie, gdy funkcja otrzyma jakiś obiekt, sprawdzana jest jego klasa i jeżeli istnieje metoda dla tego obiektu to wówczas jest ona używana. Natomiat w sytuacji, gdy dla danej klasy obiektu nie ma istniejącej metody, używana jest domyślna metoda (`.default`).

Jak widać w powyższym przykładzie, nowe klasy obiektów oraz nowe metody są tworzone, aby ułatwić pracę na innych niż domyślne strukturach danych.

7.6. Zadania

1) Stwórz trzy nowe macierze - `ma1`, `ma2`, `ma3` - składające się z trzech wierszy i czterech kolumn. Macierz `ma1` powinna zawierać wartości od 0 do 11, cała macierz `ma2` powinna składać się tylko z wartości 2, a macierz `ma3` powinna zawierać losowe wartości od 1 do 3 (w stworzeniu losowych wartości może pomóc funkcja `sample()`).
2) Wykonaj podstawowe operacje, takie jak dodawanie, odejmowanie, mnożenie i dzielenie używając macierzy `ma1` oraz `ma2`, a następnie macierzy `ma1` i `ma3`. Co jest efektem tych obliczeń? W jaki sposób działania arytmetyczne są wykonywane na macierzach w R?
3) Wydziel tylko pierwszy wiersz i ostatnią kolumnę macierzy `ma1`.
4) Znajdź wartości macierzy `ma3`, które są większe niż 2.
5) Połącz kolumnami macierz `ma1` i macierz `ma3` tworząc nowy obiekt `ma4`
6) Stwórz nową ramkę danych, `ra1`, która składa się z dwóch kolumn i trzech wierszy. Pierwsza kolumna `data` zawiera datę z dziś, wczoraj i przedwczoraj, a kolumna `miasto` zawiera nazwę miasta w którym się właśnie znajdujesz.
7) Stwórz nową ramkę danych, `ra2`, która również składa się z dwóch kolumn i trzech wierszy. Kolumna `tmin` zawiera wartości `5.3`, `4.6`, `2.9`, a kolumna `tmax` zawiera wartości `11.1`, `14.6`, `9`.
8) Połącz dwie stworzone ramki danych `ra1` i `ra2` tworząc obiekt `ra3`. Używając obiektu `ra3` wylicz średnią temperaturę dla każdego wiersza i wpisz ją w nową kolumnę `tmean`.
9) Zmień nazwę drugiej kolumny w obiekcie `ra3` na `"tmaks"`, a trzeciej na `"tsr"`.
10) Wyświetl tylko te daty dla których średnia temperatura była wyższa niż 8.

11) Stwórz nową listę, `li1`, która zawiera trzy elementy. Pierwszy element to wektor liczb od 10 do 0, drugi element to obiekt `ma4`, a trzeci element to obiekt `ra3`.
12) Wydziel z tej listy pierwszy element i nazwij go `wektor_l`.
13) Wylicz średnią wartość z kolumny `"tsr"` z trzeciego elementu listy.
14) Zamień obiekt `ramka1` utworzony w tym rozdziale na macierz. Co jest efektem zamiany klasy?

8. Powtarzanie

W sekcji 1.2 zostały wspomniane różne istniejące paradygmaty programowania. Pętle `for` czy `while` (sekcje 8.1 i 8.2) są przykładami programowania imperatywnego, gdzie program komputerowy postrzegany jest jako ciąg poleceń dla komputera. Alternatywą do tego sposobu działania jest programowanie funkcyjne, w którym rozwiązanie pewnego problemu jest oparte o użycie lub stworzenie odpowiedniej funkcji (sekcja 8.3). Język R pozwala na stosowanie zarówno paradygmatu imperatywnego jak i paradygmatu funkcyjnego[1].

8.1. Pętla for

Pęlta `for` jest jednym z najczęściej używanych wyrażeń w językach programowania[2], którego celem jest powtórzenie pewnej operacji o znaną liczbę razy.

8.1.1. Składnia

Pęlta `for` jest zbudowana z dwóch elementów: nagłówka określającego powtórzenia, oraz ciała zawierającego obliczenia.

```
for (element in wektor) {
  przetwarzanie elementu
}
```

8.1.2. Przykład działania

Zobaczmy jak działa pętla `for` na uproszczonym przykładzie zamiany wartości odległości z mil lądowych na kilometry. Nasze dane wejściowe to lista składająca się z trzech wartości - 142, 63, oraz 121. Wiemy też, że jedna mila lądowa to 1,609 kilometra.

[1] R również obsługuje paradygmat obiektowy.
[2] https://en.wikipedia.org/wiki/For_loop

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

Pętla `for` może być użyta w tym przypadku na kilka sposobów. Na początku warto zastanowić się w jaki sposób można zamienić tylko jedną wartość z powyższej listy. Wiemy, że do wybrania jednego elementu z listy służy operator `[[]]` (sekcja 7.3.4), więc przeliczenie i wyświetlenie tylko pierwszego elementu można wykonać poprzez:

```
print(odl_mile[[1]] * 1.609)
#> [1] 228
```

Teraz naszym celem jest potwórzenie tej operacji dla każdego elementu.

```
print(odl_mile[[1]] * 1.609)
#> [1] 228
print(odl_mile[[2]] * 1.609)
#> [1] 101
print(odl_mile[[3]] * 1.609)
#> [1] 195
```

W powyższym przypadku mamy tylko trzy elementy, ale jeżeli mielibyśmy takich elementów 1000 musielibyśmy powtórzyć niemal tą samą linię kodu tysiąc razy jedynie zamieniając numer elementu.

Jednym z celów programowania jest ułatwienie szybkiej powtarzalności pewnych czynności. Dlatego w tym przypadku moglibyśmy uniknąć wieloktornego pisania podobnego kodu używając pętli `for`. Ciałem tej pętli będzie sposób przeliczania i wyświetlania wartości na kilometry, ale zamiast wydzielać kolejne elementy listy (`[[1]]`, `[[2]]`, `[[3]]`), użyjemy nowego obiektu `i`. W efekcie nowe ciało pętli `for` będzie przedstawiać się jako `print(odl_mile[[i]] * 1.609)`. Kolejnym krokiem jest odpowiednie ustawienie jakie wartości będzie przyjmować `i` w kolejnych powtórzeniach.

```
for (i in 1:3) {
  print(odl_mile[[i]] * 1.609)
}
#> [1] 228
#> [1] 101
#> [1] 195
```

Powyższy nagłówek pętli `for`, `for (i in 1:3)`, określa, że nasz obiekt `i` przyjmie najpierw wartość 1, wykona obliczenie wewnątrz pętli, następnie `i` przyjmie wartość 2, znów wykona obliczenie, a na końcu `i` przyjmie wartość 3 i obliczenie zostanie wykonane po raz ostatni.

Tradycyjnie zmienna w pętli `for` nazywana jest `i`, a w przypadku zagnieżdzonych pętli druga zmienna nazywana jest `j`. Nazywanie zmiennych w ten sposób nie jest jednak obowiązkowe. W powyższym przypadku możliwe byłoby nazwanie zmiennej, np. `pomiar`: `for (pomiar in odl_mile) {...}`.

Użyty wyżej kod wykonuje nasz cel, ale wymaga od nas zawsze deklaracji dotyczącej tego jakie wartości ma przyjąć obiekt `i`. W przypadku, gdy obiekt wejściowy `odl_mile` będzie krótszy lub dłuższy niż trzy elementy, będziemy musieli ręcznie zmienić nagłówek pętli `for`. Aby tego uniknąć możemy automatycznie określić wszystkie pozycje elementów w liście `odl_mile` używając funkcji `seq_along()`. Ta funkcja zawsze wyświetli numery położenia kolejnych elementów danego wektora lub listy.

```
seq_along(odl_mile)
#> [1] 1 2 3
```

Poniższy kod nie wymaga już od nas ręcznego wprowadzania kolejnych położeń elementów wejściowej listy.

```
for (i in seq_along(odl_mile)) {
  print(odl_mile[[i]] * 1.609)
}
#> [1] 228
#> [1] 101
#> [1] 195
```

Często w takich sytuacjach używana jest konstrukcja `1:length()`, np. `1:length(old_mile)`. Zadziała ona poprawnie w powyższym przypadku, ale nie jest ona uniwersalna. Konstrukcja `1:length()` może wywołać problemy w kodzie, gdy wejściowy obiekt jest pusty. `for (i in 1:length(NULL)){...}` wykona pętlę `for` dwa razy, podczas gdy w rzeczywistości nie powinna ona zostać w ogóle wykonana. Funkcja `seq_along()` jest odporna na ten problem - `seq_along(NULL)` nie wykona pętli ani razu.

Wcześniejsze przykłady wyświetlały przeliczone na kilometry kolejne elementy listy `odl_mile`. Możliwe było zobaczenie nowych wartości, ale nie zostawały one w pamięci komputera - w efekcie nie można było wykorzystać wyników działania pętli `for` w przyszłości. Co w takim razie należy zrobić, aby wynik dało się użyć dalej? Jednym z podejść jest modyfikacja istniejącej list `odl_mile`. Poniższa pętla `for` zastępuje kolejne wartości z obiektu `odl_mile` na kilometry.

```
for (i in seq_along(odl_mile)) {
  odl_mile[[i]] = odl_mile[[i]] * 1.609
}
odl_mile
#> [[1]]
#> [1] 228
#>
#> [[2]]
#> [1] 101
#>
#> [[3]]
#> [1] 195
```

Niestety, w efekcie stracone zostały oryginalne wartości w milach lądowych.

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

Aby zostawić oryginalne wartości w milach lądowych, ale też stworzyć nowy obiekt określony w kilometrach musimy stworzyć nowy, pusty obiekt, a następnie wypełnić go wartościami. Poniżej nazwany on został `odl_km` - jest to pusta lista. Następnie kolejne wykonania pętli `for` doklejają kolejne elementy do tej listy.

```
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
#> [[1]]
#> [1] 228
#>
#> [[2]]
```

```
#> [1] 101
#>
#> [[3]]
#> [1] 195
```

Efektem jest poprawne rozwiązanie naszego problemu, ale niestety posiada ono istotną wadę - to rozwiązanie nie jest bardzo wydajne. Za każdym przejściem pętli następuje bowiem alokacja pamięci, co zabiera niepotrzebnie czas. Więcej informacji na ten temat można znaleźć w rozdziale 11.

Lepszym rozwiązaniem w takiej sytuacji jest od razu stworzenie listy, o długości zgodnej z naszym oczekiwaniem. Następnie kolejne elementy stworzonej listy są zamieniane na oczekiwane przez nas wartości.

```
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
#> [[1]]
#> [1] 228
#>
#> [[2]]
#> [1] 101
#>
#> [[3]]
#> [1] 195
```

8.1.3. Zastosowanie w funkcjach

Pętle `for`, podobnie jak wyrażenia warunkowe (sekcja 4.4), w naturalny sposób są stosowane w funkcjach. Przykładowo, możemy stworzyć nową funkcję `mile_na_km()`, która przyjmuje listę z wartościami w milach lądowych jako obiekt wejściowy, a później zwraca listę z wartościami w kilometrach.

```
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
}
```

Sprawdźmy działanie funkcji na prostym przykładzie listy z pięcioma elementami.

```
odleglosci_mile = list(0, 1, 10, 55, 160)
mile_na_km(odleglosci_mile)
#> [[1]]
#> [1] 0
#>
#> [[2]]
#> [1] 1.61
#>
#> [[3]]
#> [1] 16.1
#>
#> [[4]]
#> [1] 88.5
#>
#> [[5]]
#> [1] 257
```

Zgodnie z oczekiwaniami zero mil lądowych to również zero kilometrów, a jedna mila lądowa to 1,609 kilometra.

8.2. Pętla while

W przypadku pętli `for` znana jest liczba powtórzeń przed rozpoczęciem jej działania. Inny rodzaj pętli, pętla `while`, jest natomiast stosowany gdy nie wiadomo ile potwórzeń jest koniecznych. W efekcie pętla `while` jest bardziej elastyczna, co jest zarazem jej atutem i wadą. Bardziej elastyczne metody charakteryzuje większa liczba potencjalnych sytuacji do których mogą zostać użyte, ale w efekcie też więcej potencjalnych problemów. Pętla `while` powinna być używana tylko gdy rozwiązanie z użyciem pętli `for` nie jest możliwe.

Pętla `while` składa się z nagłówka definującego pewien warunek oraz ciała określającego operację do wykonania. Pętla ta będzie tak długo powtarzana jak długo warunek będzie spełniony - dlatego też w ciale pętli musi być jakiś mechanizm zmieniający wartość wpływającą na warunek.

```
while (warunek){
    wykonuj operację tak długo jak warunek jest spełniony
}
```

Wyobraźmy sobie poniższą sytuację. Mamy 1000 zł (obiekt `budzet`) i chcemy zainwestować te pieniądze na giełdzie w celu ich pomnożenia. Interesują nas tylko dwa scenariusze - jeden w którym tracimy całą kwotę, oraz drugi w którym udaje się nam podwoić tę kwotę. Wiemy też jedną dodatkową rzecz - losowe wahania na giełdzie mogą pozwolić nam na stratę maksymalnie 100 zł aż do zysku 100 zł każdego dnia. Poniższy kod wykonuje pętlę `while` tak długo jak obiekt `budzet` ma wartość większą od zera i mniejszą od 2000.

```
budzet = 1000
liczba_dni = 0
while(budzet > 0 && budzet < 2000){
  budzet = budzet + sample(-100:100, size = 1) # losowa strata lub zysk
  liczba_dni = liczba_dni + 1
}
```

Po jego wykonaniu możemy dowiedzieć się czy udało się nam zarobić czy też stracić całe pieniądze. Dodatkowo możemy sprawdzić ile zajęło to dni.

```
budzet
#> [1] -33
liczba_dni
#> [1] 522
```

Inne istniejące rodzaje pętli to pętla `repeat` oraz pętla `do`. Pętla `repeat` powtarza pewnien kod aż do momentu przerwania go przez użytkownika (np. użycie klawisza Esc) lub do pojawienia się komendy `break`. Działanie pętli `do` natomiast wygląda w następujący sposób: `do {wykonuj operację} while (warunek)`. Pętla `do` nie występuje w R.

Dodatkowe informacje na temat pętli for and while można znaleźć w sekcji Loops[3] książki Advanced R (Wickham, 2014)

8.3. Programowanie funkcyjne

Sprawdźmy działanie programowania funkcyjnego na dwóch przykładach. W pierwszym posiadamy listę `pomiary_f_lista` składającą się z trzech elementów - każdy z nich to wektor z trzema pomiarami temperatury w stopniach Fahrenheita.

[3] https://adv-r.hadley.nz/control-flow.html#loops

```
pomiary_f_lista = list(
  miastoA = c(61, 14, 21),
  miastoB = c(43, 52, 30),
  miastoC = c(41, 42, 33)
)
pomiary_f_lista
#> $miastoA
#> [1] 61 14 21
#>
#> $miastoB
#> [1] 43 52 30
#>
#> $miastoC
#> [1] 41 42 33
```

Naszym celem jest zamiana tych wartości na stopnie Celsjusza. Używając paradygmatu imperatywnego, moglibyśmy zastosować pętlę `for` i zastosować przeliczenie wartości dla kolejnych elementów listy. W paradygmacie funkcyjnym natomiast naszym pierwszym krokiem jest stworzenie funkcji wykonującej podstawową operację:

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

Funkcje użyte w programowaniu funkcyjnym muszą spełniać dwa warunki:

1. Wynik działania funkcji musi zależeć od obiektu wejściowego, czyli gdy dwa razy uruchomimy tą samą funkcję na tych samych danych musimy dostać ten sam wynik. Taka funkcja nie może mieć w sobie, np. elementu losowego.
2. Funkcja nie może mieć efektów ubocznych (ang. *side-effects*), czyli wykonywać jakiegoś działania w tle, jak np. wyświetlanie czy zapisywanie na dysk.

Powyższa funkcja `konwersja_f_to_c()` działa poprawnie na wektorach wartości, ale niestety nie jest w stanie zwrócić wyniku w przypadku listy, co obrazuje komunikat błędu.

```
konwersja_f_to_c(pomiary_f_lista)
#> Error in temperatura_f - 32: non-numeric argument to binary operator
```

Języki obsługujące programowanie funkcyjne posiadają jednak szereg narzędzi do przetwarzania funkcji, które zbiorczo są nazywane funkcjonałami (ang. *functional*). Funkcjonały to funkcje, które przyjmują inne funkcje jako argumenty.

W R istnieje cała rodzina funkcji poświęcona programowaniu funkcyjnemu. Oprócz najczęściej używanych, `lapply()` i `apply()`, istnieją również takie funkcje jak `sapply()`, `vapply()`, `tapply()`, `mapply()` i inne.

Jednym z podstawowych funkcjonałów w R jest `lapply()`. Funkcjonał `lapply()` przyjmuje jako pierwszy arugment wektor atomowy lub listę, a następnie przetwarza go używając funkcji podanej jako drugi argument `FUN`.

Poniżej, `lapply()` wykonuje funkcję `konwersja_f_to_c()` na kolejnych elementach listy `pomiary_f_lista` i zwraca nową listę zawierającą wyniki

```
pomiary_c_lista = lapply(pomiary_f_lista, FUN = konwersja_f_to_c)
pomiary_c_lista
#> $miastoA
#> [1]  16.11 -10.00  -6.11
#>
#> $miastoB
#> [1]  6.11 11.11 -1.11
#>
#> $miastoC
#> [1] 5.000 5.556 0.556
```

Programowanie funkcyjne można też stosować do innych klas obiektów. Poniższa ramka danych `pomiary` zawiera trzy kolumny z pomiarami temperatury dla kolejnych miast. Dla każdego miasta wykonano jeden pomiar dziennie.

```
pomiary = data.frame(
  miastoA = c(6.1, 1.4, -2.1),
  miastoB = c(4.3, 5.2, 3.0),
  miastoC = c(4.1, 4.2, 3.3)
)
pomiary
```

```
#>   miastoA miastoB miastoC
#> 1     6.1     4.3     4.1
#> 2     1.4     5.2     4.2
#> 3    -2.1     3.0     3.3
```

Naszym celem jest wyliczenie średnich - zarówno średniej wartości dla każdego miasta (kolumny) oraz średniej wartości dla każdego dnia (wiersze). Możemy to zrobić używając pętli `for`. Najpierw tworzymy pusty wektor `sr_miasto` o długości oczekiwanego wyniku, a następnie wyliczamy średnią dla kolejnych kolumn i dodajemy ją do tego wektora.

```
sr_miasto = vector("numeric", length = ncol(pomiary))
for(i in seq_len(ncol(pomiary))){
  sr_miasto[i] = mean(pomiary[, i])
}
sr_miasto
#> [1] 1.80 4.17 3.87
```

W kolejnym kroku tworzymy pusty wektor `sr_dzien` również o długości oczekiwanego wyniku, a następnie wyliczamy średnią dla kolejnych wierszy i dodajemy ją do tego wektora.

```
sr_dzien = vector("numeric", length = nrow(pomiary))
for(i in seq_len(nrow(pomiary))){
  sr_dzien[i] = mean(unlist(pomiary[i, ]))
}
sr_dzien
#> [1] 4.83 3.60 1.40
```

Alternatywą w takich przypadkach jest użycie programowania funkcyjnego, a w szczególności funkcjonału `apply()`. Oczekuje on co najmniej trzech argumentów, `X` - obiektu wejściowego którym mogą być między innymi ramki danych czy macierze, `MARGIN` określającego czy wartości będą grupowane po wierszach czy kolumnach, oraz `FUN` zawierającego używaną funkcję.

W poniższym przypadku obiektem wejściowym jest ramka danych `pomiary`, `MARGIN = 2` oznacza wyliczanie oddzielnie dla kolejnych kolumn przy użyciu zdefiniowanej funkcji `mean()`.

```
apply(pomiary, MARGIN = 2, FUN = mean)
#> miastoA miastoB miastoC
#>    1.80    4.17    3.87
```

Podobne obliczenie, ale dla kolejnych wierszy można uzyskać zamieniając argument `MARGIN` na `1`.

```
apply(pomiary, MARGIN = 1, FUN = mean)
#> [1] 4.83 3.60 1.40
```

Pakiet **purrr** oferuje ulepszone i rozszerzone narzędzia do programowania funkcyjnego (Henry and Wickham, 2019). Przykładowo, odpowiednikiem funkcji `lapply()` w pakiecie **purrr** jest funkcja `map()`. Ma ona dodatkowo kilka kolejnych wariantów, np. `map_df()` - która przyjmuje jako wejście listy, ale zwraca ramki danych, czy `map_dbl()` - która również przyjmuje listy, ale zwraca wartości zmiennoprzecinkowe.

8.4. Zadania

1) Spójrz na poniższy kod, ale nie wykonuj go. Ile razy zostanie wyświetlony tekst `"Działa!"`?

```
for (i in c(1, 2, 4, 5, 6)){
    if (i < 2 | i >= 5)
      print("Działa!")
}
```

2) Spójrz na poniższy kod, ale nie wykonuj go. Ile razy zostanie wyświetlony tekst `"Działa!"`?

```
for (i in c(1, 2, 4, 5, 6)){
  for (j in 6:3){
    if (i < 2 | i >= 5)
      print("Działa!")
  }
}
```

3) Spójrz na poniższy kod, ale nie wykonuj go. Ile razy zostanie wyświetlony tekst `"Działa!"`?

```
for (i in c(1, 2, 4, 5, 6)){
  for (j in 6:3){
    if (i < 2 & j >= 5)
      print("Działa!")
  }
}
```

4) Spójrz na poniższy kod, ale nie wykonuj go. Ile razy zostanie wyświetlony tekst `"Działa!"`?

```
for (i in c(1, 2, 4, 5, 6)){
  for (j in 6:3){
    if (i < 2 | j >= 5)
      print("Działa!")
  }
}
```

5) Spójrz na poniższy kod, ale nie wykonuj go. Ile razy zostanie wyświetlony tekst `"Działa!"`?

```
for (i in c(1, 2, 3)){
  for (j in 6:3){
    i = i + j
    if (i < 4 | i >= 9)
      print("Działa!")
  }
}
```

6) Posiadasz listę zawierającą wartości temperatury w stopniach Fahrenheita, która została przedstawiona na początku sekcji 8.3. Stwórz pętlę `for`, która zamieni te wartości na stopnie Celsjusza.

7) Rozwiąż powyższe zadanie używając również pętli `while`. Dodatkowo: spróbuj rozwiązać to zadanie używając pętli `repeat`.

8) Posiadasz listę zawierającą wartości odległości w milach lądowych, która została przedstawiona na początku sekcji 8.1.2. Użyj metod programowania funkcyjnego, żeby przeliczyć jej wartości na kilometry.

9) Posiadasz ramkę danych `pomiary` zawierającą wartości temperatury dla kolejnych miast, która została przedstawiona w sekcji 8.3. Napisz dwie funkcje używając pętli `for` - jedna, która znajdzie drugą najwyższą wartość w każdym wierszu oraz druga, która znajdzie drugą najwyższą wartość w każdej kolumnie.

10) Rozwiąż poprzednie zadanie korzystając z metod programowania funkcyjnego zamiast pętli `for`.

11) Posiadasz macierz o wymiarach czterech wierszy na sześć kolumn składającą się z kolejnych liter alfabetu.

```
m = matrix(LETTERS[1:24], ncol = 6, nrow = 4)
m
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,] "A"  "E"  "I"  "M"  "Q"  "U"
#> [2,] "B"  "F"  "J"  "N"  "R"  "V"
#> [3,] "C"  "G"  "K"  "O"  "S"  "W"
#> [4,] "D"  "H"  "L"  "P"  "T"  "X"
```

Napisz funkcję wykorzystującą pętle `for` aby określić "sąsiadów" kolejnych liter wykorzystując sąsiedztwo oparte tylko o wspólną krawędź (ang. *4-neighborhood*). Przykładowo, sąsiadem litery "A" są litery "E" i "B".

12) Przepisz powyższą funkcję, aby wykorzystywała sąsiedztwo oparte także o wspólne punkty (ang. *8-neighborhood*). Przykładowo, sąsiadem litery "A" są litery "E", "B" i "F".

9. Wczytywanie i zapisywanie plików

Języki programowania mają na celu wykonywanie wielu złożnych operacji w relatywnie krótkim czasie. Często te działania oparte są o dane zewnętrzne, np. stworzone przez człowieka, automatyczny sensor, czy jako efekt działania innego programu. Czasem też konieczne jest przekazanie lub udostępnienie wyników obliczeń dla innych osób. Celem tego rozdziału jest wprowadzenie do zagadnień związanych z określaniem położenia plików na dysku komputera, wykonywaniu działań na plikach i folderach oraz pobieraniu danych z internetu. Posiadając taką więdzę możliwe jest wczytywanie i zapisywanie danych, takich jak dane tekstowe, dane w formatach R, czy dane z arkuszy kalkulacyjnych.

9.1. Folder roboczy

Folder roboczy (ang. *working directory*) to miejsce na dysku, w którym aktualnie pracujemy. Folder roboczy można sprawdzić korzystając z funkcji `getwd()`:

```
getwd()
#> [1] "/home/travis/build/Nowosad/elp"
```

Zmienić folder roboczy można za pomocą skrótu *Ctrl+Shift+H* w RStudio (inaczej *Session -> Set Working Directory -> Choose Directory..*) lub też funkcji `setwd()`:

```
setwd("home/jakub/Documents/elp/") #unix
setwd("C:/Users/jakub/Documenty/elp/") #windows
```

Ustawienie folderu roboczego ma też miejsce przy tworzeniu nowego lub otwieraniu istniejącego projektu RStudio.

Folder roboczy jest ważny ponieważ pozwala na korzystanie z względnej ścieżki. Ścieżka względna oznacza określanie ścieżki pliku w odniesieniu do istniejącego folderu roboczego, podczas, gdy ścieżka bezwzględna opisuje pełne położenie pliku. Przykładowo, mamy plik `dane_meteo.csv`, którego pełna ścieżka to `home/jakub/Documents/elp/pliki/dane_meteo.csv`. Z poziomu R ten plik jest widoczny zarówno jako `home/jakub/Documents/elp/pliki/dane_meteo.csv`, ale też w postaci `pliki/dane_meteo.csv`[1]. Używanie ścieżek względnych jest rekomendowane, ponieważ znacząco upraszcza pracę, gdy dane/obliczenia przenosi się pomiędzy różnymi komputerami lub gdy współpracuje się z innymi osobami. Ścieżki względne są też używane w połączeniu z systemami kontroli wersji (rozdział 12).

Ścieżki folderów w systemach Windows są domyślnie rozdzielane ukośnikiem wstecznym (`\`, ang. *backslash*), jednak R pozwala także na użycie prawego ukośnika (`/`, ang. *slash*). Prawy ukośnik jest rekomendowany, ponieważ działa on zarówno na Windowsach, jak i komputerach z systemami MacOS czy Linux.

9.2. Działania na plikach i folderach

Z poziomu R możliwe jest również zarządzanie folderami i plikami na dysku. Do tworzenia nowych folderów służy funkcja `dir.create()`, np. `dir.create("dane")` stworzy nowy podfolder o nazwie `"dane"`. Sprawdzenie czy folder już istnieje możliwe jest używając funkcji `dir.exists()`, np. `dir.exists("dane")`, która zwraca wartość `TRUE` gdy folder o tej nazwie istnieje lub `FALSE` gdy takiego folderu nie ma. Do usuwania istniejących folderów służy funkcja `unlink()`. W jej przypadku konieczne jest podanie, oprócz nazwy folder do usunięcia, argumentu `recursive = TRUE`. Przykładowo, aby usunąć folder `"dane"` należy wpisać `unlink("dane", recursive = TRUE)`.

Sprawdzenie czy plik istnieje na dysku można wykonać używając `file.exist()`, a usunąć go za pomocą `file.remove()`. Obie funkcje przyjmują jako wejście wektor znakowy zawierający nazwy plików do sprawdzenia czy usunięcia.

W przypadkach, gdy konieczne jest stworzenie nowego archiwum ZIP lub rozpakowanie istniejącego pliku w tym formacie można użyć funkcji `zip()` oraz `unzip()`[2].

[1] Jeżeli poprawnie ustawiliśmy folder roboczy.

[2] Na komputerach z systemem Windows wymagane jest posiadanie zainstalowanego programu do rozpakowywania plików ZIP.

9.3. Dane internetowe

Pliki, które chcemy otworzyć nie muszą od razu znajdować się na dysku naszego komputera. Możliwe jest, między innymi, pobranie ich z poziomu R za pomocą funkcji `download.file()`[3]. Należy w niej podać adres URL pliku do pobrania, oraz nazwę pliku do zapisania.

```
download.file("https://raw.githubusercontent.com/Nowosad/elp/master/pliki/dane_meteo.csv",
              destfile = "pliki/dane_meteo_url.csv")
```

W efekcie plik `dane_meteo_url.csv` zostanie zapisany w folderze `pliki`. Funkcja `download.file()` ma też szereg dodatkowych argumentów, między innymi `method` określającą metodę pobierania danych oraz `mode` określający sposób zapisu pliku.

9.4. Wczytywanie plików tekstowych

Podstawowymi sposobami przechowywania informacji są pliki tekstowe i binarne. Przykładowo, pliki tekstowe mogą przechowywać dane w postaci tabelarycznej, a jednym z najczęściej używanych formatów tekstowych jest CSV (ang. *comma-separated values*).

Wyobraźmy sobie, że otrzymaliśmy plik tekstowy zawierający wybrane pomiary meteorologiczne dla Poznania oraz Zakopanego w roku 2017.

```
#> [1] "kod_stacji,nazwa_stacji,rok,miesiac,dzien,tavg,precip"
#> [2] "352160330,POZNAŃ,2017,1,1,1.4,0"
#> [3] "352160330,POZNAŃ,2017,1,2,0.1,0"
#> [4] "352160330,POZNAŃ,2017,1,3,0.5,4.8"
#> [5] "352160330,POZNAŃ,2017,1,4,1.5,2.3"
```

Zapisaliśmy ten plik jako `dane_meteo.csv` w podfolderze `pliki`, więc jego ścieżka względna to `"pliki/dane_meteo.csv"`. Po otworzeniu tego pliku w edytorze tekstu widzimy, że pierwszy jego wiersz zawiera nazwy kolumn, a następne wiersze to kolejne obserwacje. Dodatkowo można zobaczyć, że kolumny rozdzielane są przecinkami (`,`), natomiast wartości zmiennoprzecinkowe kropkami (`.`).

Do wczytania tego pliku możemy użyć wbudowanej w R funkcji `read.csv()`, podając w niej bezwzględną lub względną ścieżkę do pliku. W poniższych

[3] Funkcje takie jak `read.csv()` pozwalają na otworzenie pliku tekstowego bezpośrednio z adresu internetowego.

przykładzie dodatkowo ustalono argument `stringsAsFactors` na `FALSE`, dzięki czemu kolumny z tekstem nie będą zamieniane na klasę czynnikową[4].

```
meteo = read.csv("pliki/dane_meteo.csv",
                 stringsAsFactors = FALSE)
str(meteo)
#> 'data.frame':    730 obs. of  7 variables:
#>  $ kod_stacji  : int  352160330 352160330 352160330 352160330 352160330 352160330 352160330 352160330 352160330 352160330 ...
#>  $ nazwa_stacji: chr  "POZNAŃ" "POZNAŃ" "POZNAŃ" "POZNAŃ" ...
#>  $ rok         : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 ...
#>  $ miesiac     : int  1 1 1 1 1 1 1 1 1 1 ...
#>  $ dzien       : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ tavg        : num  1.4 0.1 0.5 1.5 -3.5 -8.4 -7.9 -7.7 -5.8 -5 ...
#>  $ precip      : num  0 0 4.8 2.3 0 0 2.1 0 0 0 ...
```

Oprócz funkcji `read.csv()` istnieje również funkcja o nazwie `read.csv2()`. Pierwsza z nich jest przystosowana do wczytania danych dla których separator kolumn to , a separator dziesiętny to ., druga natomiast jest używana gdy wejściowe dane mają ; jak separator kolumn i , jako separator dziesiętny.

Czasem dane tekstowe posiadają inne znaki służące jako separatory, czy też nie posiadają nazw kolumn. W takich sytuacjach można użyć funkcji `read.table()`, która zawiera cały szereg argumentów, które można dopasować, aby poprawnie wczytać dane tekstowe.

```
meteo = read.table("pliki/dane_meteo.csv",
                   sep = ",",
                   header = TRUE)
str(meteo)
#> 'data.frame':    730 obs. of  7 variables:
#>  $ kod_stacji  : int  352160330 352160330 352160330 352160330 352160330 352160330 352160330 352160330 352160330 352160330 ...
#>  $ nazwa_stacji: Factor w/ 2 levels "POZNAŃ","ZAKOPANE": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ rok         : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 ...
#>  $ miesiac     : int  1 1 1 1 1 1 1 1 1 1 ...
#>  $ dzien       : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ tavg        : num  1.4 0.1 0.5 1.5 -3.5 -8.4 -7.9 -7.7 -5.8 -5 ...
#>  $ precip      : num  0 0 4.8 2.3 0 0 2.1 0 0 0 ...
```

[4]Takie defensywne zachowanie może oszczędzić pewnych problemów w przyszłości.

R posiada kilka dodatkowych funkcji pozwalających na odczytywanie plików tekstowych, tj. `read.delim()`, `read.delim2()`, `read.fwf()`, czy `readLines()`. Funkcje `read.delim()` oraz `read.delim2()` są odpowiednikami `read.csv()` i `read.csv2()` dla plików, gdzie kolejne zmienne są oddzielane tabulatorami. Funkcja `read.fwf()` służy do odczytywania danych o ustalonej długości kolejnych zmiennych. Funkcja `readLines()` wczytuje kolejne linie z pliku tekstowego. Efektem jej działania w przeciwieństwie do poprzednich funkcji nie jest ramka danych, ale wektor tekstowy, gdzie każdy kolejny element wektora to tekst z kolejnych linii.

9.5. Zapisywanie plików tekstowych

Plik `pliki/dane_meteo.csv` zawiera pomiary ze stacji Poznań oraz Zakopane. W przypadku, gdy interesują nas tylko informacje dla Poznania możemy wydzielić odpowiednie wiersze używając funkcji `subset()` (sekcja 7.2.3).

```
meteo_pzn = subset(meteo, nazwa_stacji == "POZNAŃ")
str(meteo_pzn)
#> 'data.frame':    365 obs. of  7 variables:
#>  $ kod_stacji  : int  352160330 352160330 352160330 352160330 352160330 352160330 352160330 
#>  $ nazwa_stacji: Factor w/ 2 levels "POZNAŃ","ZAKOPANE": 1 1 1 1 1 1 1 1 1 ...
#>  $ rok         : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 ...
#>  $ miesiac     : int  1 1 1 1 1 1 1 1 1 ...
#>  $ dzien       : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ tavg        : num  1.4 0.1 0.5 1.5 -3.5 -8.4 -7.9 -7.7 -5.8 -5 ...
#>  $ precip      : num  0 0 4.8 2.3 0 0 2.1 0 0 0 ...
```

Następnie możemy zapisać obiekt `meteo_pzn` do nowego pliku używając funkcji `write.csv()`[5], poprzez podanie nazwy obiektu do zapisania oraz ścieżki zapisu wynikowego pliku. Dodatkowo możliwe jest pominięcie zapisania nazw wierszy (`row.names = FALSE`).

```
write.csv(meteo_pzn,
          file = "pliki/dane_meteo_pzn.csv",
          row.names = FALSE)
```

[5]Istnieje również jej odpowiednik `write.csv2()`.

Funkcje `read.csv()` czy `write.csv()` są domyślnie dostępne w języku R. Ich wydajność nie jest niestety najlepsza w przypadku plików tekstowych o dużej wielkości. W takich przypadkach warto użyć alternatywnych funkcji, np. `read_csv()` i `write_csv()` z pakietu **readr** (Wickham et al., 2018a) lub `fread()` i `fwrite()` z pakietu **data.table** (Dowle and Srinivasan, 2019).

9.6. Formaty R

Formaty tekstowe są bardzo uniwersalne pozwalając na odczyt, zapis czy przenoszenie danych pomiędzy różnymi komputerami, programami czy językami programowania. Mają one jednak pewne ograniczenia. Wielkość pliku tekstowego rośnie bardzo szybko wraz z liczbą elementów, a plik tekstowy nie przechowuje dodatkowych informacji specyficznych dla języków programowania. Wczytanie czy zapis dużego pliku tekstowego zabiera też relatywnie dużo czasu.

Przykładowo, chcemy aby kolumna `nazwa_stacji` była reprezentowana jako wektor czynnikowy.

```
meteo$nazwa_stacji = as.factor(meteo$nazwa_stacji)
str(meteo)
#> 'data.frame':    730 obs. of  7 variables:
#>  $ kod_stacji  : int  352160330 352160330 352160330 352160330 352160330 352160330 352160330 
#>  $ nazwa_stacji: Factor w/ 2 levels "POZNAŃ","ZAKOPANE": 1 1 1 1 1 1 1 1 1 ...
#>  $ rok         : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 ...
#>  $ miesiac     : int  1 1 1 1 1 1 1 1 1 ...
#>  $ dzien       : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ tavg        : num  1.4 0.1 0.5 1.5 -3.5 -8.4 -7.9 -7.7 -5.8 -5 ...
#>  $ precip      : num  0 0 4.8 2.3 0 0 2.1 0 0 0 ...
```

W przypadku zapisania nowego obiektu do pliku tekstowego ta informacja zostanie utracona. Alternatywnie, jeżeli chcemy używać tych danych tylko w języku R możemy zapisać je do pliku w binarnym formacie RDS. Taki zapis można wykonać używając funkcji `saveRDS()` podając nazwę obiektu do zapisu oraz ścieżkę do nowego pliku.

```
saveRDS(meteo, file = "pliki/dane_meteo.rds")
```

Taki plik będzie domyślnie mniejszy (nastąpi jego kompresja przy zapisie) niż tekstowy i będzie posiadał on wszelkie informacje o klasie tego obiektu. Ponowne wczytanie obiektu można wykonać używając funkcji `readRDS()`.

```
meteo_rds = readRDS("pliki/dane_meteo.rds")
str(meteo_rds)
#> 'data.frame':    730 obs. of  7 variables:
#>  $ kod_stacji  : int  352160330 352160330 352160330 352160330 352160330 352160330 352160330 
#>  $ nazwa_stacji: Factor w/ 2 levels "POZNAŃ","ZAKOPANE": 1 1 1 1 1 1 1 1 1 ...
#>  $ rok         : int  2017 2017 2017 2017 2017 2017 2017 2017 2017 2017 ...
#>  $ miesiac     : int  1 1 1 1 1 1 1 1 1 ...
#>  $ dzien       : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ tavg        : num  1.4 0.1 0.5 1.5 -3.5 -8.4 -7.9 -7.7 -5.8 -5 ...
#>  $ precip      : num  0 0 4.8 2.3 0 0 2.1 0 0 0 ...
```

Powyżej można zobaczyć, że została zachowana wcześniejsza zmiana - kolumna `nazwa_stacji` nadal jest reprezentowana poprzez wektor czynnikowy.

Dodatkowo możliwe jest zapisywanie wielu obiektów do jednego pliku w formacie o rozszerzeniu `.rda` a następnie ich odczytanie używając wbudowanych funkcji `save()` i `load()`.

Pakiety to zorganizowany zbiór funkcji rozszerzający możliwości R (sekcja 3.5). Istotnym elementem każdego pakietu jest jego dokumentacja, która ułatwia użytkownikom zrozumienie i wykorzystanie możliwości danego pakietu. Często, aby pokazać szczegółowo działanie danej funkcji wykorzystywane są przykładowe dane, które można dołączyć do pakietu.

Przykładowo, wraz z R domyślnie instalowany jest pakiet o nazwie **datasets**. Aby wyświetlić listę zbiorów danych w tym pakiecie można użyć funkcji `data()` i podać w niej nazwę konkretnego pakietu.

```
data(package = "datasets")
```

Wczytanie zbioru danych z wybranego pakietu odbywa się poprzez wybór nazwy zbioru (np. `"faithful"`) oraz nazwy pakietu `"datasets"`.

```
data("faithful", package = "datasets")
```

Po wykonaniu tej funkcji zbiór danych jest dostępny z poziomu R. Zawiera on ramkę danych z dwoma kolumnami opisującymi gejzer Old Faithful

(rycina 9.1): `eruptions` - czas erupcji oraz `waiting` - czas oczekiwania na kolejną erupcję.

```
head(faithful)
#>   eruptions waiting
#> 1      3.60      79
#> 2      1.80      54
#> 3      3.33      74
#> 4      2.28      62
#> 5      4.53      85
#> 6      2.88      55
```

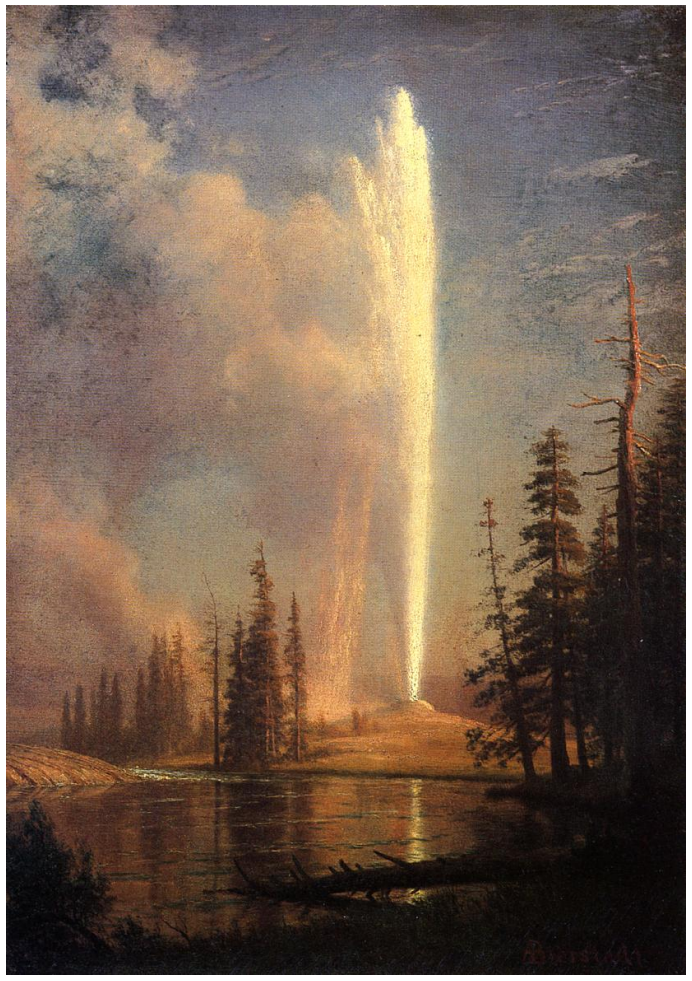
9.7. Arkusze kalkulacyjne

Powszechnym sposobem przechowywania danych tabelarycznych są arkusze kalkulacyjne tworzone w programie Microsoft Excel. Tego typu dane można wczytać do R używając pakietu **readxl** (Wickham and Bryan, 2019a). Główną funkcją tego pakietu jest `read_excel()`, przyjmująca ścieżkę pliku do wczytania. Dodatkowe argumenty tej funkcji pozwalają, np. na zdefiniowanie arkusza do wczytania (`sheet`) czy zasięgu komórek (`range`).

```
library(readxl)
meteo_z_xl = read_excel("pliki/dane_meteo.xlsx")
head(meteo_z_xl)
#> # A tibble: 6 x 7
#>   kod_stacji nazwa_stacji   rok miesiac dzien  tavg
#>        <dbl> <chr>        <dbl>   <dbl> <dbl> <dbl>
#> 1  352160330 POZNAŃ        2017       1     1   1.4
#> 2  352160330 POZNAŃ        2017       1     2   0.1
#> 3  352160330 POZNAŃ        2017       1     3   0.5
#> 4  352160330 POZNAŃ        2017       1     4   1.5
#> 5  352160330 POZNAŃ        2017       1     5  -3.5
#> 6  352160330 POZNAŃ        2017       1     6  -8.4
#> # ... with 1 more variable: precip <dbl>
```

Do zapisania ramki danych do formatu Excel można użyć funkcji `write_xlsx()` z pakietu **writexl** (Ooms, 2018)[6].

[6]Po więcej informacji otwórz plik pomocy tej funkcji `?write_xlsx()`.



Rysunek 9.1.: Obraz Alberta Bierstadta przedstawiający gejzer Old Faithful około roku 1881. Źródło: https://commons.wikimedia.org/wiki/File:Bierstadt_Albert_Old_Faithful.jpg

9.8. Inne formaty

Powyżej można było zobaczyć jaki sposób można wczytać dane z różnorodnych plików tekstowych, R, czy arkuszy kalkulacyjnych. R pozwala jednocześnie na otworzenie wielu innych formatów plików używając dodatkowych pakietów. Przykładowo, hierarchiczne formaty danych można wczytać używając pakietu **jsonline** (format `.json`) (Ooms et al., 2018) czy **xml2** (format `.xml`) (Wickham et al., 2018b), a formaty danych przestrzennych używając pakietu **sf** (dane wektorowe) (Pebesma, 2019) czy **raster**

(dane rastrowe) (Hijmans, 2019). Zestawienie zawierające dodatkowe przykłady można znaleźć pod adresem https://github.com/leeper/rio.

Powszechną sytuacją jest przechowywanie danych w różnego rodzaju bazach danych. Ma to miejsce, kiedy dane są znacznej wielkości, mają złożone relacje, czy też muszą być jednocześnie dostępne dla wielu osób. Dostęp do baz danych w R możliwy jest używając pakietu **DBI** (R Special Interest Group on Databases (R-SIG-DB) et al., 2018) wraz z dodatkowym pakietem dla konkretnego systemu bazodanowego (np. **RPostgreSQL** dla baz PostgreSQL (Conway et al., 2017) czy **RSQLite** dla baz SQLite (Müller et al., 2018)). [7]

9.9. Zadania

1) Stwórz nowy projekt RStudio o nazwie "IO". Wszystkie kolejne zadania wykonuj wewnątrz tego projektu.
2) Sprawdź jaki jest obecny folder roboczy.
3) Stwórz z poziomu R dwa nowe foldery, jeden o nazwie `"data"` i drugi o nazwie `"R"`.
4) Pobierz plik `pliki.zip` znajdujący się pod adresem https://github.com/Nowosad/elp/raw/master/. Rozpakuj go do podfolderu `"data"` i usuń pobrane archiwum z poziomu R.
5) Wczytaj dane z pliku `dane_meteo.xlsx`. Usuń z nowego obiektu pierwszą kolumnę i zapisz go jako plik w binarnym formacie RDS.
6) Wczytaj pliki `dane_meteo.csv` oraz `dane_meteo2.csv` do R. Połącz te dwa obiekty łącząc kolumny, a następnie zapisz nowy obiekt do pliku ".
7) Napisz funkcję, która przyjmuje jako argument nazwę folderu, a następnie wczytuje wszystkie pliki o rozszerzeniu `.csv` znajdujące się w tym folderze i łączy je kolumnami.
8) W pliku `list.txt` znajduje się zaszyfrowana wiadomość. Aby ją odczytać należy stworzyć odwrotność jej zawartości, tj. pierwsza linia tekstu w pliku wejściowym ma być ostatnią linią tekstu w przetworzonym obiekcie, a pierwszy znak w danej linii ma stać się ostatnim, itd. Napisz funkcję, która odszyfruje tą wiadomość.

[7] Więcej informacji o łączeniu się z bazami danych można znaleść na stronie https://db.rstudio.com/getting-started/connect-to-database.

Część II.

Narzędzia

10. Złożone funkcje

Funkcje są podstawą działania w językach programowania. Rozdział 3 wprowadził do podstawowych kwestii związanych z funkcjami - jak się używa wbudowanych funkcji oraz jak się tworzy proste nowe funkcje. Tworzenie bardziej złożonych funkcji czy też zbiorów funkcji wymaga przemyślenia tego nie tylko jak się będą one nazywać, ale też tego jak mogą one zostać użyte przez inne osoby. W tym rozdziale zostanie podanych kilka porad w jaki sposób budować funkcje przyjazne innym użytkownikom oraz w jaki sposób tworzyć odpowiednie komunikaty błędów, ostrzeżeń czy wiadomości. Dodatkowo, nastąpi także wprowadzenie do kolejnego paradygmatu programowania - programowania obiektowego.

10.1. API

Interfejs programistyczny aplikacji (ang. *application programming interface*, API) to zbiór sposobów komunikacji pomiędzy różnymi komponentami oprogramowania. Inaczej mówiąc API określa w jaki sposób następuje interakcja z kodem. Dobrze zaprojektowane API uławia zarówno rozwijanie oprogramowania, jak i jego używanie. Podstawowe elementy przemyślanego API w R obejmują nazwy funkcji, ich argumenty, oraz tzw. stabilność typu (ang. *type stability*).

Funkcje wewnątrz pojedynczego pakietu powinny być nazywane konsekwentnie używając tylko jednej konwencji nazywania (sekcja 2.4.1). Sama nazwa powinna w zwięzły sposób przekazywać jakie jest działanie funkcji. Dodatkową możliwością jest używanie w jednym pakiecie funkcji rozpoczynających się od takiego samego prefiksu. Przykładowo, większość nazw funkcji w pakiecie **landscapemetrics** rozpoczyna się od liter `lsm_`, np. `lsm_l_ent()` (Hesselbarth et al., 2019).

Podobnie należy stosować tylko jedną konwencję przy nazywaniu argumentów funkcji, a nazwy argumentów powinny być informacyjne, ale jednocześnie zwięzłe. W przypadku, gdy taki sam rodzaj danych wejściowych jest oczekiwany w różnych funkcjach, koniecznie jest aby zawsze ten argument był tak samo nazwany. Podobnie należy zadbać o spójną kolejność podobnych argumentów w funkcjach jednego pakietu.

Stabilność typu oznacza, że używając jednej klasy danych wejściowych funkcja zawsze zwróci obiekt jednej klasy. Poniższy przykład użycia funkcji

`grep()` pokazuje, że nie ma ona stabilności typu. Zalecane jest unikanie tworzenia funkcji bez stabilności typu.

```
tekst = c("kołdra", "kordła", "pościel")
grep("^[k].", x = tekst)
#> [1] 1 2
grep("^[k].", x = tekst, value = TRUE)
#> [1] "kołdra" "kordła"
```

Dodatkowym elementem API może być określenie domyślnych parametrów funkcji. Poniższa funkcja, `potegowanie()` ma na celu podnoszenie wartości wejściowego wektora (`x`) do wybranej potęgi (`w`). Domyślamy się jednak, że większość użytkowników jest zainteresowana używaniem tej funkcji do podnoszenia wartości do drugiej potęgi i dlatego też ustalamy, że domyślnie argument `w` przyjmuje wartość 2.

```
potegowanie = function(x, w = 2){
  x ^ w
}
```

W tej sytuacji, gdy użytkownik poda tylko jeden argument do funkcji `potegowanie()` to podany wektor zostanie podniesiony do kwadratu.

```
potegowanie(2)
#> [1] 4
```

Będzie to identyczne z działaniem funkcji, gdy użytkownik ręcznie zdefiniuje drugi argument jako dwa (`w = 2`).

```
potegowanie(2, w = 2)
#> [1] 4
```

W sytuacji, gdy użytkownika interesuje inna wartość `w` niż domyślna, może on ją zmodyfikować i otrzyma odpowiedni wynik.

```
potegowanie(2, w = 3)
#> [1] 8
```

10.2. Obsługa komunikatów

W sekcji 3.9 omówiliśmy trzy podstawowe rodzaje komunikatów: błędy, ostrzeżenia i wiadomości. Teraz zobaczmy jak te zaimplementować we własnych funkcjach i kiedy powinny być one użyte.

Obsługa błędów w funkcjach ma na celu ochronę użytkownika przed nieodpowiednim zachowaniem funkcji. Komunikat błędu powinien ułatwiać użytkownikowi zrozumienie problemu oraz jego rozwiązanie. Zazwyczaj komunikat błędu przyjmuje jedną z trzech form: (1) określenie problemu, np. `Argument 'x' musi być zmienną numeryczną, a nie znakową.`, (2) lokalizacja błędu, np. `Kolumna 'abc' nie istnieje w obiekcie 'y'.`, (3) porada, np. `Did you mean 'Species == "setosa"'?`. Oczywiście te wymienione formy można łączyć.

Ważne jest też, aby funkcja kończyła swoje działanie jak najszybciej po napotkaniu, np. błędnych wartości wejściowych. Żadnej użytkownik nie chce czekać na zakończenie wykonywania długiej funkcji zanim dostanie komunikat błędu. Więcej informacji o strukturze komunikatów błędów można znaleźć na https://style.tidyverse.org/error-messages.html.

Do zatrzymania działania funkcji i wyświetlenia komunikatu błędu służy `stop()`.

```
stop("To jest komunikat błędu.")
#> Error in eval(expr, envir, enclos): To jest komunikat błędu.
```

Ostrzeżenia mogą być używane w wielu różnorodnych sytuacjach, np. kiedy chcesz poinformować użytkowników o tym, że dana funkcja zostanie wygaszona lub przeniesiona do innego pakietu. Komunikaty ostrzeżenia tworzyć się używając funkcji `warning()`.

```
warning("To jest komunikat ostrzeżenia.")
#> Warning: To jest komunikat ostrzeżenia.
```

Wiadomości mają na celu poinformowanie użytkownika na temat działania pakietu lub funkcji. Są one wykorzystywane podczas wczytywania niektórych pakietów. Innym przykładem jest informowanie na temat działania funkcji w tle - pobierania danych, zapisywania do pliku, czy przeliczania cząstkowych parametrów. Do wyświetlenia wiadomości służy funkcja `message()`.

```
message("To jest komunikat wiadomości.")
#> To jest komunikat wiadomości.
```

Działanie funkcji `message()` jest zbliżone do funkcji `cat()` czy `print()`. Różni je jednak cel w jakim są użyte. Rolą funkcji `message()` jest przekazanie informacji od twórcy do użytkownika, natomiast celem funkcji tj. `cat()` jest zapytanie użytkownika w pewnej kwestii.

Przykład użycia trzech podstawowych rodzajów komunikatów można zobaczyć w poniższej funkcji `minus_1()`. Ta funkcja przyjmuje wartość numeryczną, od której odejmuje jeden, a na końcu zwraca wartość bezwzględną (`abs(x - 1)`).

```
minus_1 = function(x){
  if(is.character(x)){
    stop("Argument `x` musi być zmienną numeryczną, a nie znakową.")
  } else if(is.logical(x)){
    warning(paste("Argument `x` jest zmienną logiczną.",
                  "Czy nie chcesz użyć zmiennej numerycznej?"))
  } else {
    message("Wow. Argument `x` jest oczekiwaną zmienną numeryczną.")
  }
  abs(x - 1)
}
```

W przypadku, gdy użytkownik wprowadzi jako wejście wektor tekstowy (`if(is.character(x))`) to działanie funkcji zostanie przerwane i pojawi się odpowiedni komunikat błędu.

```
minus_1("kot")
#> Error in minus_1("kot"): Argument `x` musi być zmienną numeryczną, a nie znakową.
```

Jeżeli jako argument `x` zostanie podany wektor logiczny (`else if(is.logical(x))`) to pojawi się komunikat ostrzeżenia, ale dalsze obliczanie zostanie wykonane. W tym przypadku wartość `TRUE` zostanie najpierw zamieniona na 1 a `FALSE` na zero, następnie od tych wartości zostanie odjęte jeden, a na końcu zostaną one zamienione na wartości bezwzględne.

```
minus_1(c(TRUE, FALSE))
#> Warning in minus_1(c(TRUE, FALSE)): Argument `x`
#> jest zmienną logiczną. Czy nie chcesz użyć zmiennej
#> numerycznej?
#> [1] 0 1
```

Po wprowadzeniu wartości numerycznych do funkcji `minus_1()` pojawi się tekst wiadomości, po którym nastąpi wyliczenie kodu `abs(x - 1)`.

```
minus_1(c(1, 0, 6, -6))
#> Wow. Argument `x` jest oczekiwaną zmienną numeryczną.
#> [1] 0 1 5 7
```

Złożone funkcje opierają się o inne istniejące funkcje. W powyższym przykładzie, `minus_1()` używał, między innymi funkcji - do odejmowania czy `abs` do wyliczania wartości bezwzględnej. Czasami spodziewamy się, że wartość wprowadzona przez użytkownika może spowodować wystąpienie wewnętrznego błędu i jednocześnie wiemy jak to naprawić. W takich sytuacjach przydaje się funkcja `tryCatch()`.

R pozwala na ignorowanie wystąpienia błędu używając funkcji `try()`, ignorowanie ostrzeżeń z `suppressWarnings()` oraz wiadomości z `suppressMessages()`.

`tryCatch()` stara się uruchomić jakiś wskazany kod, a w przypadku pojawienia się błędu wykonuje alternatywne obliczenia. Można to zobaczyć na poniższym przykładzie, gdzie najpierw sprawdzona zostałaby linia `kod do uruchomienia` i dopiero gdyby ona skutkowała błędem zostałaby uruchomiona linia `wykonaj kod w przypadku wystąpienia błędu`.

```
tryCatch(
  error = function(e) {
    wykonaj kod w przypadku wystąpienia błędu
  },
  kod do uruchomienia
)
```

Działanie `tryCatch` w praktyce jest pokazane w funkcji `log_safe()`. Stara się ona wyliczyć logarytm naturalny (`log()`) z wartości argumentu `x`, a w przypadku gdyby napotkała błąd zwróci ona wartość `NA`.

```
log_safe = function(x){
  tryCatch(
  error = function(e) {
    NA
  },
  log(x)
  )
}
```

Sprawdźmy jej zachowanie na dwóch przykładach. W pierwszym oryginalna funkcja `log()` jak i nowa `log_safe()` otrzymają poprawne dane wejściowe - wektor numeryczny.

```
log(10)
#> [1] 2.3
log_safe(10)
#> [1] 2.3
```

W tym przypadku obie zwracają dokładnie taki sam wynik. Jeżeli jednak jako dane wejściowe wprowadzimy wektor znakowy to oryginalna funkcja zwróci błąd, a nasza funkcja jedynie wartość `NA`.

```
log("abecadło")
#> Error in log("abecadło"): non-numeric argument to mathematical function
```

```
log_safe("abecadło")
#> [1] NA
```

Dodatkowo istnieje funkcja `withCallingHandlers()`, która jest używana w przypadku działania na ostrzeżeniach.

10.3. Programowanie obiektowe

Programowanie obiektowe (ang. *object-oriented programming*, OOP) to jeden z najpopularniejszych paradygmatów programowania (sekcja 1.2). Polega on na definiowaniu obiektów danej klasy posiadających pewną określoną strukturę oraz zachowania.

R pozwala również na stosowanie paradygmatu obiektowego. Co więcej, w tym języku istnieje kilka różnych systemów programowania obiektowego, między innymi S3, S4 czy R6. Każdy z nich charakteryzuje inny sposób tworzenia obiektów czy ich zachowań. W tym rozdziale skupimy się na najczęściej używanego systemu S3.

Dwa najważniejsze elementy tego systemu to klasy i metody. Klasa obejmuje obiekty o podobnej strukturze, które posiadają specjalną informację o nazwie klasy. Metoda natomiast to sposób zachowania funkcji w przypadku napotkania obiektu danej klasy. Przykład metody był pokazany w sekcji 7.5, gdzie funkcja `mean()` zachowywała się różnie w zależności od klasy danych wejściowych.

10.3.1. Klasy

Poniżej stworzono nową macierz `x`, która składa się z dwóch kolumn i dwóch wierszy oraz wartości 0, 0, 2 i 3. Ma ona na celu reprezentowanie figury geometrycznej - prostokąta. W najprostszej postaci prostokąt można opisać używając czterech współrzędnych - najmniejszej wartości położenia na osi x (np., `0`), najmniejszej wartości położenia na osi y (np., `0`), największej wartości położenia na osi x (np., `2`), oraz największej wartości położenia na osi y (np., `3`).

```
x = matrix(c(0, 0, 2, 3), ncol = 2)
x
#>      [,1] [,2]
#> [1,]    0    2
#> [2,]    0    3
```

Do sprawdzenia klasy obiektu w systemie S3 służy funkcja `class()`.

```
class(x)
#> [1] "matrix"
```

W efekcie upewniamy się, że klasa naszego obiektu `x` to matrix. System S3 pozwala na prostą zmianę lub dodanie nazwy klasy używając funkcji `structure()`.

```
y = structure(x, class = "prostokat")
```

Wynikiem działania tej funkcji z argumentem `class = "prostokat"` jest nowy obiekt `y`. W momencie, gdy sprawdzimy jego klasę, okaże się że nie jest to już matrix ale prostokat.

```
class(y)
#> [1] "prostokat"
```

10.3.2. Metody

Posiadamy teraz nową klasę, `prostokat`, ale nie posiadamy do niej żadnych metod. Metoda w systemie S3 to funkcja, która działa w różny sposób w zależności od klasy danych wejściowych. Możliwe jest zarówno dodanie nowej metody do istniejącej funkcji, jak i stworzenie nowej funkcji.

W tym wypadku interesuje nas możliwość policzenia powierzchni. Możemy do tego celu stworzyć nową funkcję w systemie S3 o nazwie `powierzchnia`. Pierwszym krokiem musi być określenie, że nasza funkcja ma być oparta o system S3 używając poniższej formy.

```
powierzchnia = function(x) {
  UseMethod("powierzchnia")
}
```

Drugim krokiem jest zdefiniowanie funkcji do wyliczania powierzchni prostokąta. Określa ona najpierw długości boków a i b, a następnie wymnaża je w celu wyliczenia powierzchni.

```
powierzchnia.prostokat = function(x){
  a = x[1, 2] - x[1, 1] #wyliczenie długości boku a
  b = x[2, 2] - x[2, 1] #wyliczenie długości boku b
  a * b                 #wyliczenie powierzchni prostokąta
}
```

Nazwa powyższej funkcji wygląda jakby składała się z dwóch słów oddzielonych kropką - `powierzchnia.prostokat`. W rzeczywistości jednak nazwa funkcji to tylko `powierzchnia`, a kropka sugeruje że kolejny po niej wyraz to klasa obiektu jaki przyjmie funkcja. Jest to, innymi słowy, definicja metody. Nowa funkcja `powierzchnia` zadziała w powyższy sposób tylko w wypadku otrzymania jako dane wejściowe obiektu klasy `prostokat`.

Sprawdźmy to na dwóch przykładach - obiektu `y` (klasa `prostokat`) i `x` (klasa `matrix`).

```
y
#>      [,1] [,2]
#> [1,]    0    2
#> [2,]    0    3
#> attr(,"class")
#> [1] "prostokat"
powierzchnia(y)
#> [1] 6
```

W przypadku, gdy nasz obiekt wejściowy jest klasy `prostokat` to funkcja jest wykonywana zgodnie z metodą `powierzchnia.prostokat()`,

```
x
#>      [,1] [,2]
#> [1,]    0    2
#> [2,]    0    3
powierzchnia(x)
#> Error in UseMethod("powierzchnia"): no applicable method for 'powierzchnia' applied to an ob
```

Natomiast, gdy obiekt wejściowy będzie innej klasy to pojawi się komunikat błędu sugerujący, że nie istnieje metoda dla tej klasy pozwalająca na otrzymanie wyniku.

Dodatkowo, oprócz tworzenia metod dla każdej klasy oddzielnie możliwe jest stworzenie metody domyślnej poprzez `nazwafunkcji.default`. W przypadku, gdy dla obiektu wejściowego nie istnieje metoda to wówczas wykonywana jest metoda domyślna (`default`). Poniżej dodano metodę domyślną - w przypadku, gdy dla wejściowego obiektu nie ma metody to pojawi się poniższy komunikat błędu.

powierzchnia.default = function(x) { stop("Funkcja `powierzchnia` ma wsparcie tylko dla obiektów o klasie `prostokąt`") }

Sprawdźmy działanie domyślnej metody podając macierz jako obiekt wejściowy.

```
x
#>      [,1] [,2]
#> [1,]    0    2
#> [2,]    0    3
powierzchnia(x)
#> Error in UseMethod("powierzchnia"): no applicable method for 'powierzchnia' applied to an ob
```

10.3.3. Konstruktory

Trudno oczekiwać od użytkownika, że bez żadnych pomyłek stworzy obiekt klasy, który wymyśliliśmy, a następnie użyje funkcji `structure()`, aby dodać odpowiednią nazwę klasy. Dlatego też ważnym elementem jest stworzenie konstruktora - funkcji, której celem jest zbudowanie poprawnego obiektu naszej klasy, a w przypadku podania złych argumentów wejściowych poinformowanie użytkownika co jest nie tak.

Poniżej znajduje się konstruktor o nazwie `nowy_prostokąt()`. Przyjmuje on wartości czterech współrzędnych, a następnie wykonuje szereg sprawdzeń ich poprawności:

- Czy wszystkie argumenty są typu numerycznego?
- Czy każdy argument ma tylko jeden element?
- Czy minimalna wartość współrzędnej x jest mniejsza od maksymalnej?
- Czy minimalna wartość współrzędnej y jest mniejsza od maksymalnej?

Po tych sprawdzeniach następuje zbudowanie nowej macierzy oraz dodanie nazwy klasy.

```
nowy_prostokat = function(xmin, ymin, xmax, ymax){
  vals = c(xmin, ymin, xmax, ymax)
  if (!(is.numeric(vals))){
    stop("Wszystkie argumenty muszą być typu numerycznego")
  }
  if (!all(c(length(xmin), length(ymin), length(xmax), length(ymax)) == 1)){
    stop("Każdy z argumentów może przyjmować tylko jedną wartość")
  }
  x_range = vals[3] - vals[1]
  if (x_range <= 0){
    stop("`xmax` musi przyjmować wartość większą niż `xmin`")
  }
  y_range = vals[4] - vals[2]
  if (y_range <= 0) {
    stop("`ymax` musi przyjmować wartość większą niż `ymin`")
  }
  x = matrix(vals, ncol = 2)
  structure(x, class = "prostokat")
}
```

Sprawdźmy działanie tego konstruktora na dwóch przypadkach. W pierwszym podajmy poprawne, sprawdzone wcześniej wartości.

```
nowy_p = nowy_prostokat(0, 0, 2, 3)
nowy_p
#>      [,1] [,2]
#> [1,]    0    2
#> [2,]    0    3
#> attr(,"class")
#> [1] "prostokat"
```

Konstruktor `nowy_prostokat()` działa bez problemu, zwracając nowy obiekt `nowy_p` o klasie `prostokat`. Warto od razu zobaczyć, czy ten obiekt zadziała poprawnie w funkcji `powierzchnia()`.

```
powierzchnia(nowy_p)
#> [1] 6
```

W przypadku, gdy do konstruktora zostaną podane niepoprawne wartości wejściowe pojawi się odpowiedni komunikat błędu.

```
nowy_p2 = nowy_prostokat(7, 0, 6, 0)
#> Error in nowy_prostokat(7, 0, 6, 0): `xmax` musi przyjmować wartość większą niż `xmin`
```

10.4. Zadania

1) Bez pisania kodu, zaprojektuj API zbioru funkcji R pozwalających na tworzenie podstawowych obiektów reprezentujących podstawowe figury (np. kwadrat, prostokąt, koło, trójkąt, itd.) oraz wyliczania na ich podstawie podstawowych miar (np. obwód, pole powierzchni, itd.). Nowe API powinno obejmować nazwy funkcji, nazwy ich argumentów, istnienie lub brak domyślnych wartości argumentów, klasy obiektów wejściowych i wyjściowych z tych funkcji, itd.
2) Stwórz nową klasę obiektów w R reprezentujących trójkąty. Nazwij tą nową klasę `"trojkat"`. W jaki sposób trójkąty będą reprezentowane w tej nowej klasie? (Podpowiedź: w zależności od podjętej decyzji nowa klasa może być oparta o wektory, macierze lub ramki danych.)
3) Dodaj konstruktor pozwalający innym użytkownikom na tworzenie obiektów klasy `"trojkat"`. Zastanów się jakie powinny być wartości argumentów wejściowych i napisz wewnątrz konstruktora odpowiednie sprawdzenia używając komunikatów błędów, ostrzeżeń czy też wiadomości.

4) Stwórz metodę pozwalającą na wyliczanie powierzchni trójkąta.
5) Stwórz metodę pozwalającą na określanie współrzędnych centroidu trójkąta.

11. Analiza kodu

Programując naszym celem jest tworzenie funkcji, które są zarówno poprawne oraz wydajne (zwracają wynik szybko). W tym rozdziale przedstawione będą testy jednostkowe, które sprawdzają czy funkcje zwracają oczekiwany wynik oraz metody sprawdzające wydajność funkcji, takie jak, profiling i benchmarking.

11.1. 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 (sekcja 13.10), 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 `nowy_prostokat()` oraz `powierzchnia()` stworzonych w sekcji 10.3.

```
nowy_prostokat = function(xmin, ymin, xmax, ymax){
  if (!all(c(length(xmin), length(ymin), length(xmax), length(ymax)) == 1)){
    stop("Każdy z argumentów może przyjmować tylko jedną wartość")
  }
  vals = c(xmin, ymin, xmax, ymax)
  if (!(is.numeric(vals))){
    stop("Wszystkie argumenty muszą być typu numerycznego")
  }
  x = matrix(vals, ncol = 2)
  structure(x, class = "prostokat")
```

```
}
powierzchnia = function(x) {
  UseMethod("powierzchnia")
}
powierzchnia.prostokat = 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, 2019b).

```
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 = nowy_prostokat(0, 0, 6, 5)
expect_length(powierzchnia(nowy_p), 1)
```

Jeżeli wynik ma długość jeden to wówczas nic się nie stane. 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))
#> Error: `nowy_prostokat(1, 2, 3, 6)` did not throw an error.
```

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.

11.2. Profiling

Istnieją trzy podstawowe reguły optymalizacji kodu[1]:

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)
```

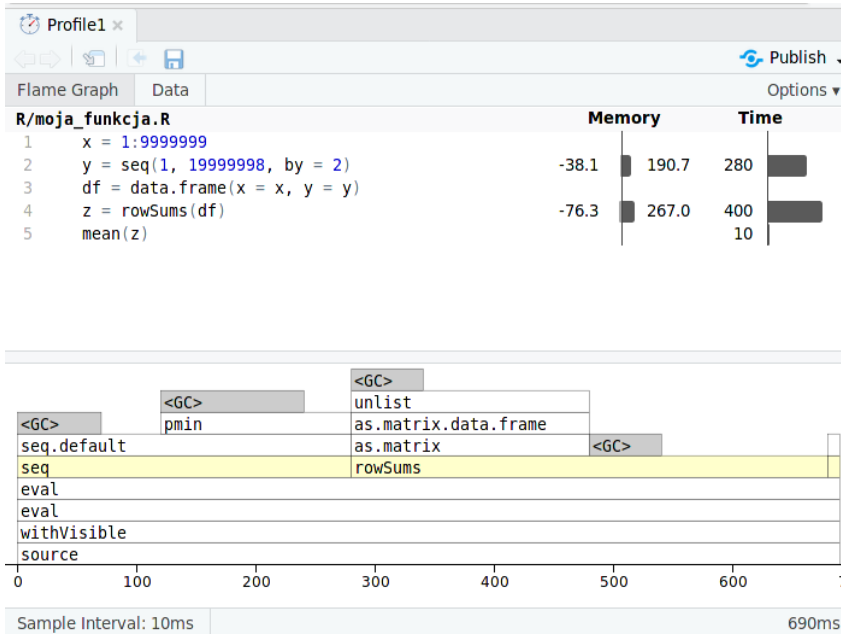
[1] http://www.moscowcoffeereview.com/programming/the-3-rules-of-optimization/

```
df = data.frame(x = x, y = y)
z = rowSums(df)
mean(z)
```

Profilowanie kodu R można wykonać używając funkcji `profvis()` z pakietu **profvis** (Chang and Luraschi, 2018). 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 (rycina 11.1).



Rysunek 11.1.: Zrzut ekranu przedstawiający wynik działania funkcji profvis().

Czas wykonania tego przykładu wyniósł sumarycznie 690ms. Pierwsza linia tworząca obiekt `x` została wykonana bardzo szybko - poniżej mierzalnego progu. Stworzenie obiektu `y` w drugiej linii zajęło ok. 280ms. Trzecia

linia została wykonana również w czasie poniżej mierzalnego 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.

11.3. 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
#>   0.612   0.000   0.612
```

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.

W sekcji 8.1.2 stworzyliśmy kilka wersji pętli `for` pozwalającej na przeliczanie wartości z mil lądowych na kilometry. Pierwsza z nich, tutaj zdefiniowana jako funkcja `mi_do_km1`, tworzy pusty wektor o długości 0, do którego następie 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, 2019), która wykonuje funkcje 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
#>   <bch:expr>           <bch:> <bch:>     <dbl> <bch:byt>
#> 1 mi_do_km1(odl_mile) 1.62us 1.77us   493286.     117KB
```

```
#> 2 mi_do_km2(odl_mile)  1.2us  1.3us   726804.     221KB
#> # ... with 1 more variable: `gc/sec` <dbl>
```

Efektem 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
#>   <bch:expr>            <bch> <bch:>     <dbl> <bch:byt>
#> 1 mi_do_km1(odl_mile2) 450ms  457ms      2.19     382MB
#> 2 mi_do_km2(odl_mile2) 773us  802us   1051.      78.2KB
#> # ... with 1 more variable: `gc/sec` <dbl>
```

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 17.52 operacji na sekundę, przy aż 21.69 operacji na sekundę funkcji `mi_do_km2`. Dodatkowo, funkcja `mi_do_km1` potrzebowała aż kilka tysięcy (!) razy więcej pamięci operacyjnej niż `mi_do_km2`.

```
#> Running with:
```

```
#>       x
#> 1    10
#> 2   100
#> 3  1000
#> 4 10000
#> Warning: Some expressions had a GC in every iteration;
#> so filtering is disabled.
#> # A tibble: 8 x 7
#>   expression       x      min   median `itr/sec`
#>   <bch:expr>   <dbl> <bch:tm> <bch:tm>     <dbl>
#> 1 mi_do_km1(l)    10   3.98us    4.5us 206897.
#> 2 mi_do_km2(l)    10   1.84us      2us 468805.
#> 3 mi_do_km1(l)   100  65.82us  70.47us  13555.
#> 4 mi_do_km2(l)   100   8.45us   9.52us  99277.
#> 5 mi_do_km1(l)  1000   4.24ms   4.48ms    222.
#> 6 mi_do_km2(l)  1000  76.11us  80.15us  12014.
#> 7 mi_do_km1(l) 10000 416.33ms 416.97ms      2.40
#> 8 mi_do_km2(l) 10000 782.39us 806.89us   1110.
#> # ... with 2 more variables: mem_alloc <bch:byt>,
#> #   `gc/sec` <dbl>
```

11.4. Zadania

1) Korzystając z wiedzy z rozdziału 8 dotyczącej pętli `for`, napisz funkcję `gdzie_naj()`, która przyjmuje na wejściu macierz z wartościami numerycznymi, wylicza sumę wartości dla każdego wiersza, a następnie zwraca numer wiersza z najwyższą sumą wartości. Przykładowe dane wejściowe do tej funkcji to:

```
set.seed(2019-05-08)
mat = matrix(c(sample(1:10, size = 25, replace = TRUE)),
             ncol = 5, nrow = 5)
mat
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    6    1    2    4    6
#> [2,]    9    9    9    5    5
#> [3,]    2    6    4    8   10
#> [4,]    4    6    1    2    5
#> [5,]    2   10    6    4    9
```

1) Dodaj do powyższej funkcji odpowiednie komunikaty błędów czy ostrzeżeń (sekcja 10.2), pojawiające się w zależności od rodzaju wprowadzonych danych wejściowych.

2) Napisz testy jednostkowe sprawdzające (1) czy błędy pojawiają się w odpowiednich sytuacjach oraz (2) czy funkcja zwraca prawidłową wartość.

3) Użyj metod profilowania kodu w celu sprawdzenia czasów wykonywania kolejnych linii kodu. Działanie której linii kodu zabiera najwięcej czasu?

4) Stwórz funkcję `gdzie_naj2()`, której cel jest taki sam jak funkcji `gdzie_naj()`, ale zamiast pętli `for` jej działanie oparte jest o funkcję `rowSums()`.

5) Używając pakietu **bench** porównaj czas działania funkcji `gdzie_naj()` i `gdzie_naj2()`. Która z nich jest szybsza? Która z nich zużywa mniej zasobów?

12. Kontrola wersji

Systemy kontroli wersji to narzędzia pozwalające na zapamiętywaniu 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.[1] Współcześnie najbardziej popularnym systemem kontroli jest Git, któremu będzie poświęcona reszta tego rozdziału. Inne popularne systemy kontroli wersji to Concurrent Versions System (CVS), Mercurial czy Subversion (SVN).

12.1. 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.[2] 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

[1] https://en.wikipedia.org/wiki/Comparison_of_version-control_software#History_and_adoption

[2] Nie w oknie konsoli R.

zmian, a także ich wielkość może powodować znaczne powiększanie się repozytorium.[3]

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.

12.1.1. Konfiguracja systemu Git

Kolejnym krokiem po instalacji systemu Git[4] 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"
```

12.1.2. Repozytorium

Podstawowym z nich jest repozytorium (ang. *repository*, często określane skrótowo jako repo). Jest to folder, który przechowuje wszystkie pliki i foldery w ramach jednego projektu.[5] Dodatkowo wewnątrz repozytorium znajduje się ukryty folder `.git`, który zawiera informację 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*)). Istnieje wiele serwisów internetowych pozwalających na tworzenie, przechowywanie i edycję repozytoriów zdalnych, między innymi GitHub[6] (przybliżony w sekcji 12.2), GitLab[7], czy BitBucket[8].

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

[3] Między innymi z tego powodu internetowe serwisy kontroli wersji posiadają ograniczenia dotyczące wielkości plików. Przykładowo, GitHub ogranicza wielkość pojedynczych plików do 100MB.
[4] Instrukcje dotyczące instalacji Gita znajdują się we wstępie książki.
[5] W kontekście R, warto o tym myśleć jako o projekcie RStudio.
[6] https://github.com/
[7] https://gitlab.com/
[8] https://bitbucket.org/

12.1.3. 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.

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

12.1.4. 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
```

12.1.5. Zatwierdzanie zmian

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

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

12.1.6. 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 `master`.

```
# 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.

```
# utworzenienie 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 (sekcja 12.1.3) i zatwierdzić (sekcja 12.1.5). Teraz można powrócić do głównej gałęzi (`master`) i dołączyć zmiany stworzone w innej gałęzi.

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

12.1.7. 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
```

12.1.8. 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
```

12.1.9. 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 zaszły w zdalnym repozytorium na lokalnym komputerze.

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

12.2. 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.

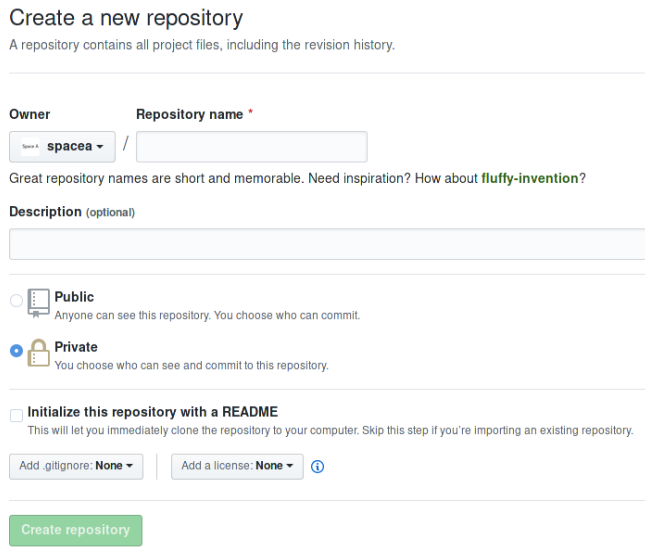
Repozytoria połączone są z kontami użytkowników (np. https://github.com/Nowosad to moje konto, gdzie "Nowosad" oznacza nazwę użytkownika) lub organizacjami (np. https://github.com/r-spatialecology to konto organizacji "r-spatialecology"). Pod adresem https://github.com/join można założyć nowe konto użytkownika.

12.2.1. 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 (rycina 12.1).

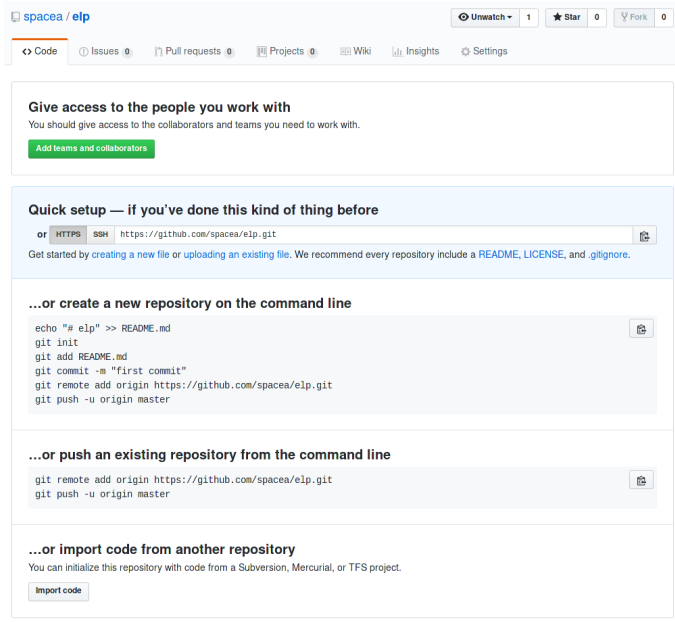
Rysunek 12.1.: Ikona tworzenia nowego repozytorium GitHub.

W kolejnym oknie (rycina 12.2) 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.



Rysunek 12.2.: Okno tworzenia nowego repozytorium GitHub.

Po wybraniu potwierdzenia (*Create repository*) utworzone zostanie nowe, puste repozytorium (rycina 12.3).



Rysunek 12.3.: Nowe, puste repozytorium GitHub.

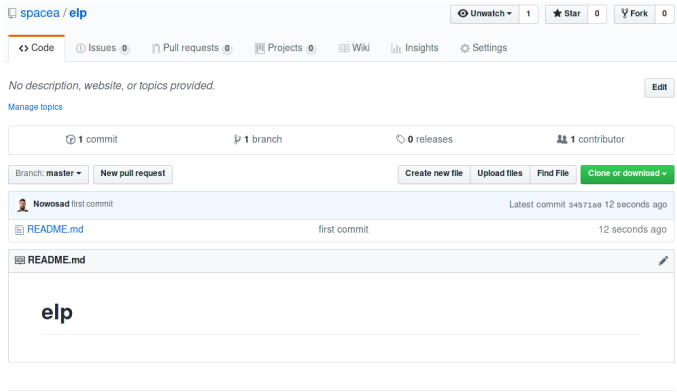
Okno pustego repozytorium przedstawia cztery główne drogi pozwalające na dodanie zawartości:

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 12.3 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.

12.2.2. Repozytorium GitHub

Wygląd okna repozytorium zmienia się po dodaniu pierwszej zawartości (rycina 12.4).



Rysunek 12.4.: Repozytorium GitHub po dodaniu zawartości.

Teraz możliwe jest podejrzenie 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.

12.2.3. Dodatkowe możliwości GitHub

W prawym górnym rogu okna repozytorium (rycina 12.4) 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. Sprawy (ang. *issues*) to miejsce, gdzie twórcy mogą zapisywać swoje listy zadań dotyczące danej aplikacji, a użytkownicy mogą zgłaszać błędy czy propozycje ulepszeń. Zapytania aktualizacyjne są tworzone, np. w przypadku, gdy lokalnie zmieniliśmy zawartość repozytorium innego użytkownika[9] i chcemy zaproponować żeby nasza zmiana została dołączona do oryginalnego repozytorium. W takiej sytuacji często opiera się to o (1) stworzenie rozwidlenia (ang. *fork*), (2) pobranie rozwidlenia jako lokalne repozytorium, (3) edycja lokalnego repozytorium, (4) zatwierdzenie zmian i wysłanie ich do zdalnego repozytorium (rozwidlenia), (5) zaproponowanie zapytania aktualizacyjnego.

Możliwe jest również łączenie możliwości serwisu GitHub z innymi serwisami internetowymi, takimi jak Travis CI[10], Codecov[11], Gitter[12] i wiele innych[13].

12.3. Kontrola wersji w RStudio

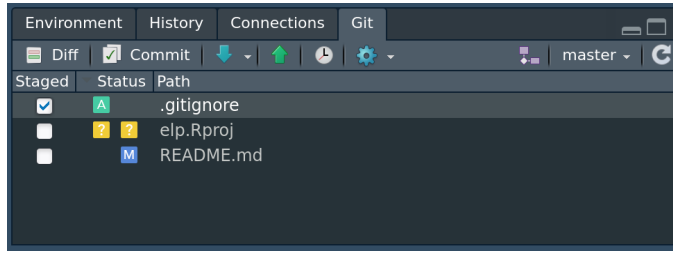
RStudio posiada wbudowane, uproszczone graficzne wsparcie dla systemu Git. Istnieje też szereg programów, których głównym celem jest ułatwienie pracy z systemem Git. Nazwane są one klientami Git, wśród których można wymienić GitKraken[14] i Sourcetree[15].

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) 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 - patrz sekcja 12.1.7), który jednocześnie jest lokalnym repozytorium Git. Dodatkowo, w RStudio pojawi się nowy panel "Git" (rycina 12.5).

[9] Może to być zarówno dodanie nowej możliwości, naprawienie błędu w kodzie, czy nawet poprawienie literówki w dokumentacji.
[10] https://travis-ci.org/
[11] https://codecov.io/
[12] https://gitter.im/
[13] https://github.com/marketplace
[14] https://www.gitkraken.com/
[15] https://www.sourcetreeapp.com/



Rysunek 12.5.: 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).[16] 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 (więcej informacji można znaleźć w sekcji 12.1.3).

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; sekcja 12.1.4) oraz zatwierdzić zmiany (prawy panel; sekcja 12.1.5). Kolejne, strzałki w dół i górę, oznaczają odpowiednio aktualizowanie zmian (sekcja 12.1.9) i wysyłanie zmian (sekcja 12.1.8). 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 (sekcja 12.1.6).

12.4. 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.

[16] Możliwe są też inne sytuacje, np. czerwona ikona z literą R sugerująca zmianę nazwy pliku.

12.4.1. Nowy projekt

Preferowanym 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. Taki sposób został opisany na początku sekcji 12.3.

W momencie, gdy posiadamy ustawione zarówno lokalne jak i zdalne repozytorium możliwe jest rozpoczęcie pracy. Teraz można tworzyć nowe oraz edytować istniejące pliki. 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 opisująca zmiany jakie zaszły, oraz ją zatwierdzić klikając przycisk *Commit*. Zalecane jest, aby powyższą czynność wykonywać nawet wiele razy dziennie.

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.

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. Tutaj zalecane jest najpierw kliknięcie ikony aktualizowania zmian (strzałka w dół), aby upewnić się, że posiadamy aktualną wersję repozytorium, a następnie 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.

Dalej praca polega na powtarzaniu tych czynności:

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

12.4.2. Istniejący projekt

Czasami posiadasz już jakiś istniejący projekt, 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.

12.5. 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.

```
>>> git push
To https://github.com/YOU/REPO.git
 ! [rejected]        master -> master (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.

Z racji popularności 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 et al., 2019) oraz rozdział Git and GitHub[17] książki R packages (Wickham, 2015). Do ogólnego wprowadzenia do systemu Git może posłużyć darmowa książka online

[17] https://r-pkgs.org/git.html

Pro Git[18] (Chacon, 2014), której kilka pierwszych rozdziałów jest również dostępna w języku polskim. Git jest również bardzo popularnym tematem na serwisie stackoverflow, gdzie można znaleźć pytania i odpowiedzi na różnorodne tematy z nim związane[19]. Więcej odnośników do materiałów związanych z sytemem Git i serwisem GitHub można znaleźć na stronach pomocy GitHub[20].

12.6. Zadania

1) Skonfiguruj system Git podając swoją nazwę użytkownika oraz adres email. Sprawdź czy nazwa została dodana używając komendy `git config --global user.name` oraz czy dodany został adres email używając `git config --global user.email`.
2) Stwórz nowe konto użytkownika lub zaloguj się na swoje istniejące konto GitHub. Utwórz nowe publiczne repozytorium o nazwie "test".
3) Połącz zdalne repozytorium "test" z nowym projektem RStudio. Sprawdź czy w RStudio pojawił się panel Git, a następnie w tym panelu dodaj pliki `.gitignore` i `test.Rproj` do repozytorium Git poprzez odhaczenie odpowiednich pól w kolumnie *Staged*. Kliknij ikonę *Commit* i wpisz wiadomość "Dodano pliki .gitignore i test.Rproj" w pole po prawej stronie. Zatwierdź tą wiadomość, a następnie prześlij te zmiany do repozytorium zdalnego.
4) Sprawdź stronę internetową zawierającą zdalne repozytorium "test". Czy zaszły na niej jakieś zmiany od poprzedniego wejścia? Przejrzyj jakie dodatkowe opcje pojawiły się na stronie tego repozytorium.
5) W projekcie "test" w RStudio stwórz nowy plik `README.md`. Do tego pliku wstaw zdanie poniższy tekst:

```
# test

To jest moje pierwsze repozytorium!
```

Dodaj ten plik do repozytorium Git, napisz odpowiedni komunikat, zatwierdź zmiany i prześlij je do repozytorium zdalnego. Sprawdź stronę internetową zawierającą zdalne repozytorium "test". Czy zaszły na niej jakieś zmiany od poprzedniego wejścia?

6) Z poziomu strony internetowej swojego repozytorium "test" edytuj plik `README.md`. Możesz to zrobić klikając na nazwę tego pliku, a następnie na ikonę ołówka w prawym górnym roku okna. Dodaj do

[18] https://git-scm.com/book/pl/v2
[19] https://stackoverflow.com/questions/tagged/git
[20] https://help.github.com/en/articles/git-and-github-learning-resources

niego kolejną linię `Edytowałem plik z poziomu GitHub.` oraz napisz odpowiedni komunikat poniżej tego okna i zawierdź zmiany (zielony przycisk `Commit changes`). Sprawdź stronę internetową zawierającą zdalne repozytorium "test". Czy zaszły na niej jakieś zmiany od poprzedniego wejścia?

7) Wróć do swojego projektu RStudio. Zobacz jak wygląda lokalny plik `README.md` - powinien on nadal zawierać wcześniej wprowadzony tekst.

```
# test

To jest moje pierwsze repozytorium!
```

Kliknij w ikonę aktualizowania zmian (strzałka w dół). Zobacz jak teraz wygląda lokalny plik `README.md`. Co się w nim zmieniło?

13. Pakiety

Pakiety są powszechnie wykorzystywane podczas pracy z językiem R. Celem sekcji 3.5 było wprowadzenie do tego czym one są, jak się je instaluje oraz dołącza. Najważniejszą tam informacją było, że pakiety są zorganizowanymi zbiorami funkcji. Oznacza to, że nie tylko posiadamy pewną liczbę stworzonych funkcji, ale także są one ułożone w pewien ustalony sposób. Funkcje w pakietach posiadają też swoją dokumentację (jej struktura została przedstawiona w sekcji 3.4) czy przykładowe dane. Pakiety, oprócz swojej unikalnej nazwy, posiadają również informacje o swojej wersji, autorach, zależnościach i licencji.

Informacje w tym rozdziale powinny pozwolić na stworzenie podstawowego pakietu R. Istnieje jednak wiele dodatkowych aspektów i kwestii w tym temacie, które zostały tutaj wspomniane pobieżnie lub pominięte. W celu poznania i zrozumienia złożnych aspektów tworzenia pakietów R cennymi źródłami wiedzy może być książki R packages[1] (Wickham, 2015) oraz rOpenSci Packages: Development, Maintenance, and Peer Review[2] (rOpenSci et al., 2019). Dodatkowo, w niektórych przypadkach pomocna może być oficjalna dokumentacja Writing R Extensions[3] (R Core Team, 2019b).

13.1. Nazwa pakietu

Nazwa nowego pakietu musi spełniać kilka wymagań: składać się tylko ze znaków ASCII[4], cyfr i kropek, mieć co najmniej dwa znaki oraz zaczynać się od litery i nie kończyć się kropką (R Core Team, 2019b). Ważne jest również myślenie o nazwie pakietu tak jak o nazwach funkcji (sekcja 2.4) - 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 et al., 2018), który sprawdza przy wybrana nazwa nie jest już zajęta oraz czy nie ma ona jakiegoś niepożądanego przez nas znaczenia.

[1] https://r-pkgs.org
[2] https://ropensci.github.io/dev_guide/
[3] https://cran.r-project.org/doc/manuals/R-exts.html#Creating-R-packages
[4] https://en.wikipedia.org/wiki/ASCII

13.2. 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 and Bryan, 2019b), 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.[5]

```
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".

13.3. Rozwijanie pakietu

Rozwój pakietu R może opierać się na kilku poniższych krokach:

1. Tworzenie/modyfikowanie kodu
2. Używanie funkcji `devtools::load_all()`, która dodaje nowe/zmodyfikowane funkcje do R
3. Sprawdzenie czy funkcja działa zgodnie z oczekiwaniami na kilku przykładach

[5] Funkcja również `create_packages()` sama tworzy nowy folder, jeżeli on wcześniej nie istniał.

4. Dodanie testów jednostkowych (sekcja 13.10) na podstawie stworzonych przykładów
5. Modyfikacja wersji oprogramowania
6. Powtórzenie powyższych czynności

13.4. Tworzenie i dokumentacja funkcji

W sekcji 3.8 stworzyliśmy nową funkcję `konwersja_temp()` przeliczającą temperaturę ze stopni Fahrenheita na stopnie Celsjusza.

```
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.

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
```

13.5. 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
```

```
Encoding: UTF-8
LazyData: true
RoxygenNote: 6.1.1
```

Tytuł pakietu (`Title:`) w jednym krótkim zdaniu (sloganie) określa do czego służy ten pakiet.[6] Składa się on ze słów rozpoczynających się z dużej litery.

Wersja pakietu (`Version:`) pozwala jego użytkownikom na zobaczenie, czy korzystają z aktualnej wersji pakietu. Zalecanym sposobem 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`.

`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 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).[7] 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")
)
```

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

[6] Tytuły pakietów można znaleźć, np. w panelu "Packages" w RStudio.

[7] Pełną listę dostępnych ról można znaleźć pod adresem http://www.loc.gov/marc/relators/relaterm.html.

GPL. Pierwsza z nich, CC0 oznacza przekazanie zawartości pakietu do domeny publicznej[8] 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 (`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.
```

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

13.6. 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 `::`.[9]

Dodatkowo każda zależność z zewnętrznym pakietem musi być określona w pliku DESCRIPTION. Jest to możliwe używając wpisów `Imports:` oraz `Suggests:`, przykładowo:[10]

[8] https://creativecommons.org/publicdomain/zero/1.0/deed.pl

[9] Istnieją również inne możliwości, np. użycie znaczników `@import` lub `@importFrom`.

[10] Istnieją również inne wpisy, takie jak `Depends:`, `LinkingTo:`, czy `Enhances:`.

```
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.

13.7. 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 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.

13.8. 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 et al., 2019).

13.9. 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.

Plik `README.Rmd` można stworzyć za pomocą funkcji `usethis::use_readme_rmd()`. W efekcie będzie się on znajdować się w głównym folderze pakietu. Ten plik powinien zawierać:[11]

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

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 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.

Zarówno plik `README.Rmd`, jak i winieta wymaga użycia odpowiedniej składni - używany jest tam tzw. język znaczników RMarkdown. Języki znaczników opierają się o założenie, że pewne znaki w pliku tekstowym mają specjalne znaczenie, które po przetworzeniu pliku wyświetla je 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 określane 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`.

[11] Dodatkowe elementy to oznaki (ang. *badges*) pokazujące, np. status pakietu, liczbę jego pobrań i wiele innych.

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()`.

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.

13.10. Wbudowane testy

Sekcja 11.1 pokazywała w jaki sposób tworzyć testy jednostkowe dla funkcji, w celu sprawdzenia czy ich działanie jest zgodne z naszymi oczekiwaniami. Takie testy 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 (zobacz sekcję 11.1). Przykładowy plik `tests/testthat/test-powierzchnia.R` może wyglądać w ten sposób:

```
nowy_p = nowy_prostokat(0, 0, 6, 5)
```

```
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()`[12]. W efekcie wyświetlone zostaną wszystkie testy i zostanie wskazane, które z nich się nie powiodły i należy je poprawić.

13.11. Publikowanie pakietów

Nowo utworzony pakiet w R można od razu umieścić na wybranym serwisie internetowym wspierającym kontrolę wersji takim jak GitHub, GitLab, czy BitBucket (rozdział 12), gdzie nazwa repozytorium będzie identyczna jak nazwa pakietu. Dodatkowo, gdy napisaliśmy plik `README.md` użytkownicy mogą dowiedzieć się do czego ten pakiet służy, jak go zainstalować i użyć w podstawowy sposób. Teraz konieczna jest promocja tego pakietu w sytuacji, gdy chcemy zainteresować inne osoby jego użyciem. Taka promocja może odbywać się poprzez ogłoszenie stworzenia tego pakietu na Twitterze używając hasztagu `#rstats`, czy też napisaniu wpisu na blogu opisującego ten pakiet.

Dodatkowo w R istnieje możliwość prostego stworzenia stron internetowych dla wybranego pakietu używając pakietu **pkgdown** (Wickham and Hesselberth, 2019). Przykład takiej strony można zobaczyć pod adresem https://pkgdown.r-lib.org/index.html. Stworzenie strony pakietu wymaga jedynie wywołania funkcji `pkgdown::build_site()` wewnątrz pakietu R. W efekcie 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 GitHub Pages, i określić Source jako “master branch /docs folder”.

[12]Testy są też automatycznie uruchamiane podczas sprawdzania pakietu (13.7)

13.12. Zadania

1) Stwórz szkielet nowego pakietu R nazwającego się **konwerter**.

2) Napisz funkcję `mil_do_km()` służącą do odległości z mili lądowych na kilometry i zapisz ją jako `R/mil_do_km.R`.

3) Uzupełnij dokumentację funkcji `mil_do_km()` zawierającą tytuł funkcji, opis funkcji, opis jej parametrów, format danych wyjściowych oraz kilka przykładów.

4) Uzupełnij opis pakietu (plik `DESCRIPTION`) poprzez podanie jego nazwy, tytułu, wersji, autora, opisu i licencji. Zastanów się nad z każdą z tych opcji. Jak powinny one wyglądać, aby potencjalny użytkownik zrozumiał do czego służy ten pakiet?

5) Sprawdź czy pakiet działa używając funkcji `devtools::check()`. Postaraj się naprawić wszystkie komunikaty błędów, ostrzeżeń i notatek jeżeli się pojawią. Zainstaluj pakiet **konwerter**.

6) Stwórz nowe repozytorium w serwisie GitHub nazywające się `konwerter`. Połącz projekt RStudio zawierający pakiet **konwerter** z tym repozytorium w serwisie GitHub (12.4.2). Prześlij wszystkie pliki na zdalne repozytorium. Sprawdź czy widzisz wszystkie pliki w repozytorium GitHub.

7) Stwórz nowy plik `README.Rmd` i dodaj do niego informacje o nazwie pakietu, jego zastosowaniu oraz w jaki sposób można go zainstalować. Dodaj również jeden przykład użycia tego pakietu. Przetwórz uzupełniony plik `README.Rmd` na `README.md` używając ikony "Knit" w RStudio lub funkcji `rmarkdown::render()`. Wyślij te dwa nowe pliki do zdalnego repozytorium. Czy widzisz jakąś zmianę w repozytorium GitHub?

8) Dodaj infrastrukturę do testów jednostkowych używając funkcji `use_testthat()`, a następnie stwórz nowy plik `tests/testthat/test-mil_do_km.R`. Wewnątrz tego pliku dodaj kilka testów jednostkowych sprawdzających, czy wynik działania funkcji `-mil_do_km()` jest zgodny z oczekiwaniami. Sprawdź czy wszystkie z testów dają odpowiedni wynik używając `devtools::test()`.

9) Stwórz stronę internetową pakietu **konwerter** używając funkcji `build_site()` z pakietu **pkgdown**. Aktywuj tę stronę internetową wewnątrz zakładki setting na stronie repozytorium `konwerter` na GitHub. Sprawdź czy strona wyświetla się zgodnie z oczekiwaniami.

10) Dodaj do tego pakietu drugą funkcję `konwersja_temp()`, która przyjmuje trzy argumenty - `x`, `z`, `na`. Pierwszy argument `x` to wektor numeryczny oznaczający temperaturę w dowolnej jednostce. Drugi argument `z` przyjmuje wartość tekstową określającą w jakiej jednostce jest obiekt

`x`, może to być `"Celsjusz"`, `"Fahrenheit"`, lub `"Kelvin"`. Trzeci argument `z` przyjmuje wartość tekstową określającą w jakiej jednostce ma być wynik działania tej funkcji, może to być `"Celsjusz"`, `"Fahrenheit"`, lub `"Kelvin"`. Funkcja `konwersja_temp()` przyjmuje temperaturę (`x`) w podanej jednostce (`z`) i przelicza ją do innej wybranej skali (`do`).

11) Sprawdź na kilku przykładach czy ta funkcja działa zgodnie z oczekiwaniami. Następnie dodaj dokumentację do funkcji `konwersja_temp()`.

12) Napisz kilka testów jednostkowych do funkcji `konwersja_temp()`. Sprawdź czy dają one poprawny wynik. Sprawdź cały pakiet używając `devtools::check()` i zainstaluj go poprzez `devtools::install()`.

13) Zaktualizuj opis pakietu (np. zmień wersję pakietu). Przebuduj stronę internetową pakietu i prześlij wszystkie zmiany na zdalne repozytorium.

14) Stwórz nową winietę do pakietu **konwerter** nazywającą się `wprowadzenie`. Dodaj do niej krótki opis tego co robi ten pakiet, a następnie przedstaw przykład użycia funkcji `mil_do_km()` oraz funkcji `konwersja_temp()`.

14. Podsumowanie

Nie jest możliwe, aby jedna książka wyczerpująco pokazywała wszystkie elementy języka programowania i podawała wszelkie jego możliwości i zastosowania. Jest to szczególnie nieosiągalne w przypadku takiego języka jak R, który posiada ogromny zbiór pakietów, oraz społeczność, która używa ten język na wiele sposobów. Celem tego rozdziału jest wskazanie co można zrobić dalej na podstawie uzyskanej wiedzy i umiejętność z tej książki.

14.1. Grafika

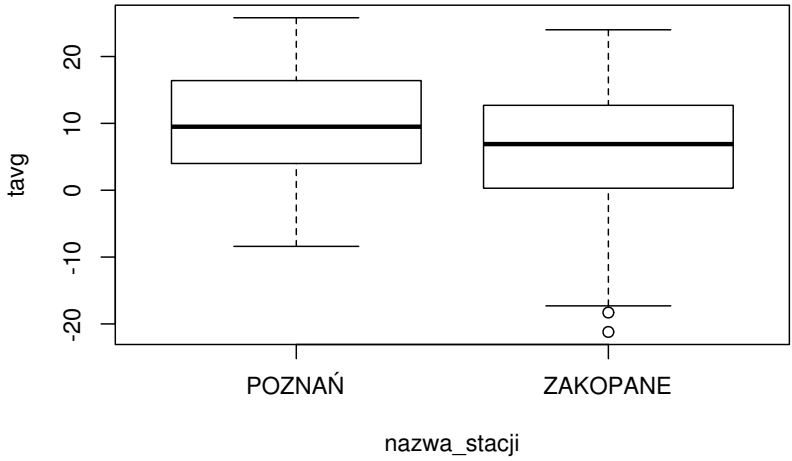
Jedną z najczęściej wymienianych zalet R są jego rozbudowane narzędzia do tworzenia wykresów. Możemy to zobaczyć na poniższym przykładzie danych meteorologicznych dla Poznania i Zakopanego z roku 2017.

```
met = read.csv("https://github.com/Nowosad/elp/raw/master/pliki/dane_meteo.csv",
               stringsAsFactors = FALSE)
head(met)
#>   kod_stacji nazwa_stacji  rok miesiac dzien tavg
#> 1  352160330       POZNAŃ 2017       1     1  1.4
#> 2  352160330       POZNAŃ 2017       1     2  0.1
#> 3  352160330       POZNAŃ 2017       1     3  0.5
#> 4  352160330       POZNAŃ 2017       1     4  1.5
#> 5  352160330       POZNAŃ 2017       1     5 -3.5
#> 6  352160330       POZNAŃ 2017       1     6 -8.4
#>   precip
#> 1    0.0
#> 2    0.0
#> 3    4.8
#> 4    2.3
#> 5    0.0
#> 6    0.0
```

Wewnątrz obiektu `met` znajdują się kolumny `tavg` (określająca średnią dobową temperaturę powietrza w stopniach Celsjusza) oraz `nazwa_stacji` ("POZNAŃ" lub "ZAKOPANE"). Do porównania wartości temperatury po-

między stacjami może posłużyć wykres pudełkowy, stworzony przy pomocy funkcji `boxplot()` (rycina 14.1). Poniżej zdefiniowano, która zmienna ma zostać zwizualizowana (`tavg`) w podziale na jakie grupy (`nazwa_stacji`) z jakiego zbioru danych (`met`)[1].

```
boxplot(tavg ~ nazwa_stacji, data = met)
```



Rysunek 14.1.: Przykład wykresu utworzonego przy pomocy funkcji boxplot().

Powyższy wykres może być zmodyfikowany używając dodatkowych argumentów (np. `main` dodający tytuł czy `col` zmieniający kolor pudełek) czy też dodatkowych funkcji pozwalających na dodanie legendy (funkcja `legend`) czy też tekstu (funkcja `text`). Inne dostępne wbudowane funkcje do tworzenia wykresów to, między innymi, `hist()` czy `barplot()` budujące histogramy oraz wykresy słupkowe.

Najbardziej elastyczną funkcją do tworzenia wykresów w R jest `plot()`. Domyślnie, gdy użytkownik poda wartości numeryczne dla argumentów `x` i `y` pozwala ona tworzyć wykresy punktowe. Jej zachowanie i wynik będzie jednak inne w zależności od tego jakiej klasy będzie obiekt wejściowy, przykładowo inaczej wyświetlony zostanie model liniowy, efekt grupowania hierarchicznego, czy też wynik testu statystycznego.

[1]Kod `tavg ~ nazwa_stacji` można inaczej odczytać jako `tavg` w zależności od `nazwa_stacji`.

Oprócz wbudowanych w R funkcji graficznych, istnieje też szereg dodatkowych pakietów służących do wizualizacji danych. Wśród nich najpopularniejszym jest **ggplot2** (Wickham et al., 2019a). Ten pakiet jest implementacją założeń zawartych w książce Grammar of Graphics (Wilkinson, 2005). Główną funkcją tego pakietu jest `ggplot()`, która przyjmuje dane wejściowe w postaci ramki danych. Wewnątrz tej funkcji następuje wywołanie kolejnej funkcji `aes`, gdzie definiowane są kolejne kolumny, które mają być wyświetlone na osiach wykresów oraz określają kolor, kształt, wielkość i inne elementy. Kolejnym krokiem jest określenie typu wykresu poprzez połączenie poprzedniej funkcji (używając operatora `+`) z jedną z wielu funkcji rozpoczynających się od `geom_`. Przykładowo, do stworzenia wykresu pudełkowego służy `geom_boxplot()` (rycina 14.2).

```
library(ggplot2)
ggplot(met, aes(nazwa_stacji, tavg)) + geom_boxplot()
```

Rysunek 14.2.: Przykład wykresu utworzonego z użyciem pakietu ggplot2.

Pełna dokumentacja pakietu **ggplot2** znajduje się na stronie http://docs.ggplot2.org.

14.2. Analiza danych

R jest jednym z języków programowania najczęściej używanych w analizie danych[2]. Jest to wynikiem szeregu przyczyn, w tym dużej liczby wbudowanych w R funkcji statystycznych oraz graficznych. Dodatkowo, ramka danych, jeden z podstawowych obiektów w R, może być utożsamiany z arkuszem kalkulacyjnym czy tabelą z bazy danych - najpopuarniejszych form przechowywania różnorakich danych. Ta forma obiektu, złożonego z kolumn (zmienne) i wierszy (obserwacje), jest reprezentacją, która uławia czyszczenie, przetwarzanie i analizowanie danych.

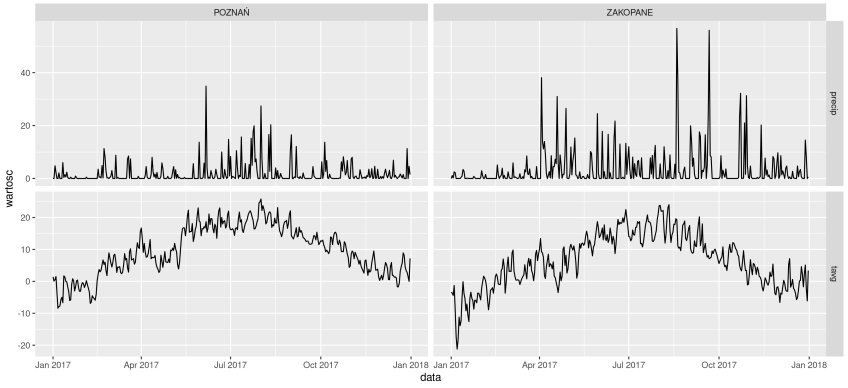
Inną przyczyną popularności R do analizy danych jest grupa pakietów zbiorczo określana jako *tidyverse*. Jest to spójny zbiór pakietów pozwalających na wykonywanie kolejnych czynności analizy danych. Na samym początku obejmuje to pakiety poświęcone wczytywaniu danych w różnych formatach, w tym poznane w rozdziale 9 pakiety **readr** (Wickham et al., 2018a) oraz **readxl** (Wickham and Bryan, 2019a). Kolejnym krokiem jest porządkowanie danych, polegające, na przykład, na zmianie struktury ramki danych gdzie wartości jakiejś zmiennej stają się nazwami kolumn. W tym etapie można użyć pakiet **tidyr** (Wickham and Henry, 2019).

Dane w odpowiedniej postaci można następnie przetwarzać, np. tworzyć nowe zmienne na postawie przeliczania już istniejących czy też wyliczać ich podsumowania używając pakietu **dplyr** (Wickham et al., 2019b). Tak przetworzone dane następnie są często wizualizowane używając pakietu **ggplot2** (Wickham et al., 2019a) lub też w ich oparciu budowane są modele[4]. Ma to na celu zrozumienie posiadanych danych oraz zależności czy zjawisk które opisują.

W ramach grupy pakietów **tidyverse** często stosuje się operator `%>%` (ang. *pipe*) z pakietu **magrittr** (Bache and Wickham, 2014). Pozwala on na łączenie kilku oddzielnych funkcji w jedno zapytanie. Działanie tego operatora polega na tym, że wynik jednej działania jednej funkcji staje się automatycznie pierwszym argumentem w kolejnej funkcji (rycina 14.3).

```
library(magrittr)
readxl::read_excel("https://github.com/Nowosad/elp/raw/master/pliki/dane_meteo.xlsx") %>%
  tidyr::gather(key = "zmienna", value = "wartosc", tavg:precip) %>%
  dplyr::mutate(data = as.Date(paste(rok, miesiac, dzien, sep = "-"))) %>%
  ggplot2::ggplot(ggplot2::aes(data, wartosc)) +
  ggplot2::geom_line() +
  ggplot2::facet_grid(zmienna~nazwa_stacji, scale = "free_y") +
```

[2]Analiza danych często jest określana również jako data science[3].
[4]https://github.com/tidymodels/tidymodels



Rysunek 14.3.: Przykład wyniku użycia pakietów z grupy tidyverse.

Pełne wprowadzenie do koncepcji *tidyverse* można znaleźć w książce R for Data Science[5] (Wickham and Grolemund, 2016).

Zrozumienie zależności czy zjawisk jest bardzo rzadko ostatnim etapem - równie istotne jest przekazanie tych wyników wybranej grupie odbiorców w odpowiedni sposób. Do tego celu może posłużyć R Markdown (jego podstawy zostały opisane w sekcji 13.9) R Markdown pozwala na tworzenie dokumentów w różnych formatach (html, pdf, docx, itd.), prezentacji, stron internetowych, książek[6] i wiele innych. Po szczegołowe instrukcje jak używać tego języka warto zajrzeć do książki R Markdown: The Definitive Guide[7] (Xie et al., 2018).

14.3. Inne zastosowania

Wcześniejsze dwie sekcje pokazywały bardzo szerokie zastosowania R - analizować czy wizualizować można zarówno dane o temperaturze powietrza jak i wyniki wyborów prezydenckich. W związku z czym, w R istnieje także znacząca liczba pakietów stworzonych do bardziej specjalistycznych i szczegółowych celów. Można to zobaczyć przeglądając tzw. *task views* - listy pakietów zaagregowane według podobnej tematyki znajdujące się pod adresem https://cran.r-project.org/web/views/. Obejmuje to bardzo szeroki przekrój tematów - od list poświęconych projektowaniu prób klinicznych, poprzez przetwarzanie języka naturalnego, skończywszy na ekonometrii i analizach finansowych[8].

[5] https://r4ds.had.co.nz/

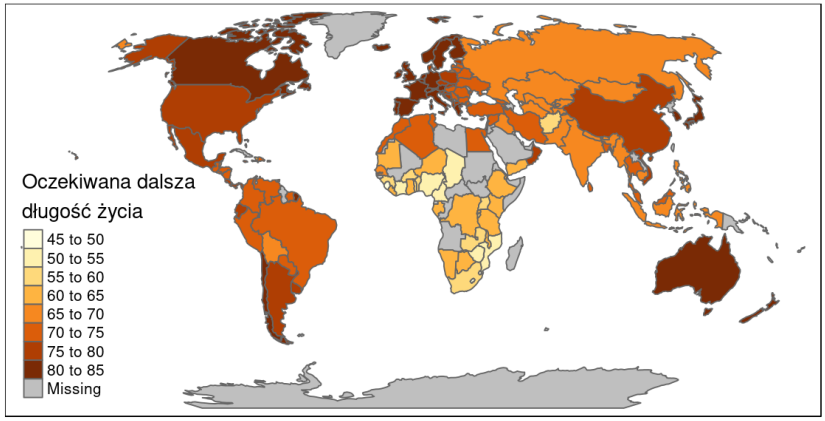
[6] Ta książka również powstała używając R Markdown.

[7] https://bookdown.org/yihui/rmarkdown/

[8] Dodatkowo istnieje specjalne repozytorium Bioconductor poświęcone pakietom R dotyczącym zagadnień bioinformatycznych.

Wsród tych list znajduje się także jedna poświęcona analizie danych przestrzennych. Opisuje ona, między innymi, takie pakiety jak **sf**, pozwalający na wczytywanie, przetwarzanie i zapisywanie danych wektorowych czy **tmap** ułatwiający tworzenie map. Na poniższym przykładzie następuje dołączenie tych pakietów oraz wczytanie zbioru danych `World` zawierającego poligony krajów na świecie i podstawowe informacje o nich. Dalej następuje dodanie tych danych do wyświetlenia i wybór odwzorowania przestrzennego używając funkcji `tm_shape()`, po czym te dane są wyświetlone w postaci poligonów (funkcja `tm_polygons()`), gdzie kolory poligonów wynikają z ich wartości w kolumnie `life_exp` a tytuł legendy jest wybrany przez nas (rycina 14.4).

```
library(sf)
library(tmap)
data(World)
tm_shape(World, projection = "robin") +
    tm_polygons(col = "life_exp",
                title = "Oczekiwana dalsza \ndługość życia")
```



Rysunek 14.4.: Przykład działania pakietu tmap.

Podstawy działania na danych przestrzennych zawiera książka Geocomputation with R[9] (Lovelace et al., 2019).

[9] https://geocompr.robinlovelace.net/

14.4. Programowanie w R

Wcześniejsze sekcje opisywały różne obszary zastosowań R, ale nie pokazywały w jaki sposób rozwijać umiejętności programowania w tym języku. Najprostszym sposobem jest używanie R jak najczęściej. Nauka języka programowania przebiega wówczas naturalnie - wraz ze znaleziskiem rozwiązania kolejnego problemu czy rozwiązaniem kolejnego zadania.

Często jednak, nie jesteśmy w stanie stwierdzić czy ten sposób rozwiązania jest optymalny, lub też napotykamy sytuacje w których nie wiemy jak się do nich odnieść. Wówczas szczególnie istotna jest inna umiejętność - czytania kodu innych osób[10]. Większość pakietów R jest otwartoźródłowych - ich kod jest dostępny online i każda chętna osoba ma do niego dostęp[11]. Kod pakietów R można, między innymi, znaleźć w serwisie GitHub. Wszystkie pakiety znajdujące się w repozytorium CRAN można znaleźć pod adresem https://github.com/cran. Inną możliwością jest samodzielne wyszukanie kodu pakietu używając wyszukiwarki GitHub - https://github.com/search.

Przykładowo, pod adresem https://github.com/karthik/wesanderson znajduje się kod źródłowy pakietu **wesanderson** (Ram and Wickham, 2018). Ten pakiet zawiera funkcje tworzące palety kolorystyczne inspirowane filmami reżysera Wesa Andersona[12]. Kod R będący podstawą działania tego pakietu znajduje się w folderze `R/`[13]. Dodatkowo, niektóre pakietu zawierają kod z innych języków programowania (np. C lub C++), który wymaga wcześniejszej kompilacji. Taki kod znajduje się w folderze `src/`.

14.5. Co dalej?

Programowanie to nie tylko pisanie kodu. Obejmuje to też wiele innych czynności, takich jak stosowanie optymalnych algorytmów czy narzędzi programistycznych. Istnieje wiele książek poświęconych kwestii algorytmów, wsród których najbardziej popularne to Introduction to Algorithms (Cormen et al., 2009), The Algorithm Design Manual (Skiena, 2008) czy Algorithms (Sedgewick and Wayne, 2011). Podstawowym narzędziem programistycznym jest program do pisania kodu. Może to być zarówno prosty edytor tekstu, taki jak Notepad++[14], Sublime Text[15], lub Atom[16] czy

[10] Read the Source, Luke.
[11] Dostępny jest także kod źródłowy samego języka R. Można go znaleźć pod adresem https://github.com/wch/r-source.
[12] https://en.wikipedia.org/wiki/Wes_Anderson
[13] Szczególnie `R/colors.R`.
[14] https://notepad-plus-plus.org/
[15] https://www.sublimetext.com/
[16] https://atom.io/

też bardziej złożone zintegrowane środowisko programistyczne (IDE). O ile narzędzia z tej pierwszej grupy są uniwersalne to w przypadku wyboru IDE warto zdecydować się na zintegrowane środowisko programistyczne odpowiednie dla używanego języka programowania[17].

Programowanie często obejmuje pracę w zespole. Wówczas jednym ze sposobów dbania o jakość tworzonego produktu może być inspekcja kodu (ang. *code review*). Polega ona na tym, że zmiany naniesione w kodzie są przekazywane innej osobie, która sprawdza go pod kątem błędów, spójności, stylu, zgodności z isniejącymi rozwiązaniami, itd. Po inspekcji twórca kodu może dostać informację zwrotną, co jest dobre, a co wymaga poprawy. W efekcie, z jednej strony wyjściowy produkt jest lepszej jakości, a z drugiej strony programista uczy się i polepsza swoje umiejętności.

Niezależnie od używanego języka istnieje również szereg narzędzi, których znajomość ułatwia lub czasem nawet umożliwia pracę. Wśród nich można wyróżnić znajomość linii komend i jej możliwości (Kross, 2017) oraz języka SQL służącego do tworzenia, edycji i zarządzania relacyjnymi bazami danych (Beighley, 2007; Forta, 2013).

Innym kierunkiem działań może być nauka kolejnego języka programowania - najlepiej takiego, którego główne zastosowanie różni się od R. Może to być przykładowo język kompilowany, taki jak C, C++ lub Rust, którego efektem będzie bardziej wydajny program. Co ważne, kod napisany w tych językach można łączyć z kodem R. R posiada wbudowany interfejs do używania kodu napisanego w C (rozdział 5 z dokumentacji Writing R Extensions[18] (R Core Team, 2019b)), łączenie kodu napisanego w C++ ułatwia znacząco pakiet **Rcpp** (Eddelbuettel et al. (2019); więcej informacji w rozdziale "Rewriting R code in C++"[19] książki Advanced R (Wickham, 2014)), a wskazówki dotyczące łączenia kodu Rust można znaleźć w repozytorium https://github.com/r-rust/hellorust. W efekcie użytkownik może korzystać z interaktywności R, wykonując dowolne linie kodu, ale część z nich może używać wydajniejszych funkcji napisanych w jezykach kompilowanych. Alternatywną drogą może być nauka języków używanych do tworzenia i rozwijania aplikacji internetowych, w tym JavaScript czy PHP.

Pomimo już znaczącej historii, języki programowania nadal mają wiele nowego do zaoferowania. Nieustannie następuje ich ewolucja - dodawane są nowe możliwości, zmieniane są istniejące funkcje, czy też następuje poprawa wydajności. Tworzone są również pakiety, moduły, czy biblioteki implementujące nowe pomysły, czy też ulepszające i rozszerzające dostępne oprogramowanie. W efekcie typowy kod napisany w danym języku kilka

[17] https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments

[18] https://cran.r-project.org/doc/manuals/R-exts.html#System-and-foreign-language-interfaces

[19] https://adv-r.hadley.nz/rcpp.html

lat temu może się różnić od tego napisanego dziś. Powstaje też ciągle wiele nowych języków, z których tylko niewielka część zdobywa szersze grono użytkowników. Te języki często wprowadzają nowe podejścia i koncepcje, które później mają bezpośredni wpływ na zmiany w istniejących językach. Języki programowania są też stosowane coraz częściej w wielu codzienne używanych sprzętach, w tym samochodach czy lodówkach (ang. *internet of things*, IOT).

Powodzenia w dalszej przygodzie z programowaniem!

Bibliografia

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2019). *rmarkdown: Dynamic Documents for R*. R package version 1.12.

Bache, S. M. and Wickham, H. (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5.

Beighley, L. (2007). *Head First SQL: Your Brain on SQL–A Learner's Guide*. " O'Reilly Media, Inc.".

Biecek, P. (2014). *Przewodnik Po Pakiecie R*. Oficyna Wydawnicza GIS.

Bryan, J., the STAT 545 TAs, and Hester, J. (2019). *Happy Git and GitHub for the useR*.

Burns, P. (2012). *The R Inferno*. Lulu.com.

Chacon, S. (2014). *Pro Git*. Apress.

Chang, W. and Luraschi, J. (2018). *profvis: Interactive Visualizations for Profiling R Code*. R package version 0.3.5.

Conway, J., Eddelbuettel, D., Nishiyama, T., Prayaga, S. K., and Tiffin, N. (2017). *RPostgreSQL: R Interface to the 'PostgreSQL' Database System*. R package version 0.6-2.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT press.

Czernecki, B. (2018). *Metody Przetwarzania Danych Meteorologicznych w Języku Programowania R*.

Dowle, M. and Srinivasan, A. (2019). *data.table: Extension of 'data.frame'*. R package version 1.12.2.

Eddelbuettel, D., Francois, R., Allaire, J., Ushey, K., Kou, Q., Russell, N., Bates, D., and Chambers, J. (2019). *Rcpp: Seamless R and C++ Integration*. R package version 1.0.1.

Forta, B. (2013). *Sams Teach Yourself SQL in 10 Minutes*. Pearson Education.

Friedl, J. E. (2006). *Mastering Regular Expressions*. " O'Reilly Media, Inc.".

Gagolewski, M. (2016). *Programowanie w Jezyku R.* Wydawnictwo Naukowe PWN.

Ganz, C., Csárdi, G., Hester, J., Lewis, M., and Tatman, R. (2018). *available: Check if the Title of a Package is Available, Appropriate and Interesting.* R package version 1.0.2.

Gillespie, C. and Lovelace, R. (2016). *Efficient R Programming: A Practical Guide to Smarter Programming.* " O'Reilly Media, Inc.".

Gries, P., Campbell, J., and Montojo, J. (2017). *Practical Programming: An Introduction to Computer Science Using Python 3.6.* Pragmatic Bookshelf.

Grolemund, G. (2014). *Hands-On Programming with R: Write Your Own Functions and Simulations.* " O'Reilly Media, Inc.".

Guzdial, M. and Ericson, B. (2016). *Introduction to Computing and Programming in Python.* Pearson.

Henry, L. and Wickham, H. (2019). *purrr: Functional Programming Tools.* R package version 0.3.2.

Hesselbarth, M. H., Sciaini, M., Nowosad, J., and Hanss, S. (2019). *landscapemetrics: Landscape Metrics for Categorical Map Patterns.* R package version 1.1.

Hester, J. (2019). *bench: High Precision Timing of R Expressions.* R package version 1.0.2.

Hester, J., Csárdi, G., Wickham, H., Chang, W., Morgan, M., and Tenenbaum, D. (2019). *remotes: R Package Installation from Remote Repositories, Including 'GitHub'.* R package version 2.0.4.

Hijmans, R. J. (2019). *raster: Geographic Data Analysis and Modeling.* R package version 2.9-5.

Kross, S. (2017). *The Unix Workbench.*

Lovelace, R., Nowosad, J., and Muenchow, J. (2019). *Geocomputation with R.* Chapman and Hall/CRC Press.

Müller, K., Wickham, H., James, D. A., and Falcon, S. (2018). *RSQLite: 'SQLite' Interface for R.* R package version 2.1.1.

Nowosad, J. (2019). *Geostatystyka w R.* Space A, Poznan.

Ooms, J. (2018). *writexl: Export Data Frames to Excel 'xlsx' Format.* R package version 1.1.

Ooms, J., Temple Lang, D., and Hilaiel, L. (2018). *jsonlite: A Robust, High Performance JSON Parser and Generator for R.* R package version 1.6.

Pebesma, E. (2019). *sf: Simple Features for R.* R package version 0.7-4.

Peng, R. D., Kross, S., and Anderson, B. (2017). *Mastering Software Development in R*.

R Core Team (2019a). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

R Core Team (2019b). *Writing R Extensions*. R Foundation for Statistical Computing.

R Special Interest Group on Databases (R-SIG-DB), Wickham, H., and Müller, K. (2018). *DBI: R Database Interface*. R package version 1.0.0.

Ram, K. and Wickham, H. (2018). *wesanderson: A Wes Anderson Palette Generator*. R package version 0.3.6.

rOpenSci, Anderson, B., Chamberlain, S., Krystalli, A., Mullen, L., Ram, K., Ross, N., Salmon, M., and Vidoni, M. (2019). rOpenSci Packages: Development, Maintenance, and Peer Review.

Sedgewick, R. and Wayne, K. (2011). *Algorithms (4th Edition)*. Addison-Wesley Professional.

Skiena, S. S. (2008). *The Algorithm Design Manual*. Springer Science & Business Media.

Wickham, H. (2014). *Advanced R*. Chapman and Hall/CRC.

Wickham, H. (2015). *R Packages: Organize, Test, Document, and Share Your Code*. " O'Reilly Media, Inc.".

Wickham, H. (2019a). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.4.0.

Wickham, H. (2019b). *testthat: Unit Testing for R*. R package version 2.1.1.

Wickham, H. and Bryan, J. (2019a). *readxl: Read Excel Files*. R package version 1.3.1.

Wickham, H. and Bryan, J. (2019b). *usethis: Automate Package and Project Setup*. R package version 1.5.0.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., and Yutani, H. (2019a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. http://ggplot2.tidyverse.org, https://github.com/tidyverse/ggplot2.

Wickham, H., François, R., Henry, L., and Müller, K. (2019b). *dplyr: A Grammar of Data Manipulation*. R package version 0.8.0.1.

Wickham, H. and Grolemund, G. (2016). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. " O'Reilly Media, Inc.".

Wickham, H. and Henry, L. (2019). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. http://tidyr.tidyverse.org, https://github.com/tidyverse/tidyr.

Wickham, H. and Hesselberth, J. (2019). *pkgdown: Make Static HTML Documentation for a Package*. https://pkgdown.r-lib.org, https://github.com/r-lib/pkgdown.

Wickham, H., Hester, J., and Francois, R. (2018a). *readr: Read Rectangular Text Data*. R package version 1.3.1.

Wickham, H., Hester, J., and Ooms, J. (2018b). *xml2: Parse XML*. R package version 1.2.0.

Wilkinson, L. (2005). *The Grammar of Graphics*. Springer.

Xie, Y. (2018). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.9.

Xie, Y. (2019). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.22.

Xie, Y., Allaire, J. J., and Grolemund, G. (2018). *R Markdown: The Definitive Guide*. Chapman & Hall.