Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego

Kierunek studiów:
Inżynieria danych

Michał Pajęcki

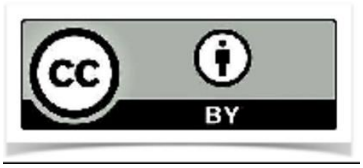
Materiały dydaktyczne do przedmiotu

PROGRAMOWANIE I ANALIZA DANYCH W R

opracowane w ramach realizacji Projektu
„**Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki**”
FERS.01.05-IP.08-0234/23

Kielce, 2025

Spis treści



Wstęp

We współczesnym świecie, w którym informacja jest jednym z najcenniejszych zasobów, kompetencje w zakresie pozyskiwania, przetwarzania i analizy danych nabierają fundamentalnego znaczenia. Kierunek ***inżynieria danych*** na Politechnice Świętokrzyskiej kształci specjalistów, którzy powinni zarówno rozumieć heterogeniczne dane, jak i być w stanie skutecznie, rzetelnie i z wykorzystaniem odpowiednich narzędzi wydobyć z nich wiedzę. Jednym z takich narzędzi jest **język R** – wszechstronny i niezwykle potężny język programowania, stworzony szczególnie z myślą o przetwarzaniu i analizie danych, czy też uczeniu maszynowym.

Celem kursu pn. ***Programowanie i analiza danych w R*** jest nie tylko nauczenie podstaw składni **języka R** i obsługi środowiska IDE **RStudio Desktop**, lecz także trening umiejętności analitycznego myślenia oraz praktycznego rozwiązywania problemów związanych z danymi – m.in. w zakresie ich pozyskiwania, przetwarzania, analizy oraz wizualizacji. Przedmiot ten realizowany jest na czwartym semestrze studiów – zakłada się więc, iż Student posiada już fundamentalną wiedzę z zakresu matematyki, statystyki i informatyki, w tym także w zakresie podstaw programowania.

Przedmiot koncentruje się na zaprezentowaniu Studentowi składni języka R, zaczynając od zagadnień podstawowych, a na średniozaawansowanych kończąc. Zakres tematyczny jest jednak ograniczony do wybranych zagadnień, które wydają się być szczególnie istotne z perspektywy ***inżynierii danych***. Należy jednak mieć świadomość, że język R to potężne i rozbudowane narzędzie, oferujące znacznie szersze możliwości niż te, które są omawiane w ramach niniejszych zajęć. Kurs ten stanowi więc swoiste wprowadzenie i punkt startowy do dalszego rozwijania kompetencji w zakresie pracy z językiem R. Być może narzędzie to będzie także wykorzystywane na innych przedmiotach w dalszym toku studiów.

Niniejsze materiały dydaktyczne obejmują wybrane zagadnienia z zakresu języka R (teoria + przykłady), które zarówno stanowią poszerzenie treści podstawowych realizowanych w trakcie trwania kursu ***Programowanie i analiza danych w R***, jak też będą przydatne Studentowi w aspekcie rozszerzenia umiejętności w kontekście przyszłej pracy z danymi. Wiele różnorodnych zagadnień z tych materiałów wykracza poza ramy tego kursu – szczególnie z racji ograniczonego czasu nie będą one poruszane na zajęciach. Ponadto zagadnienia typowo programistyczne zostały pominięte, gdyż były one przedmiotem wcześniejszych kursów.

1. Wybrane aspekty języka R

Język programowania R jest szeroko wykorzystywany w analizie danych, statystyce i uczeniu maszynowym. Powstał, by ułatwić profesjonalistom pracę z danymi. Wśród jego zalet można wyróżnić m.in. łatwość nauki, bogatą społeczność użytkowników oraz ogromną liczbę bibliotek i narzędzi umożliwiających przetwarzanie i analizę danych. Choć język R początkowo powstał myślą o analizie danych, to z biegiem czasu jego możliwości znacznie się poszerzyły i dziś R może być wykorzystywany także w szerszych zastosowaniach programistycznych.

Przy tworzeniu podrozdziałów 1.2.3 – 1.3.5 korzystano z pomocy programu R, dokumentacji R ("Rdocumentation", 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN ("The Comprehensive R Archive Network", 2025) i jego podstron oraz pozycji literaturowych (Biecek, 2018; Gągolewski, 2014; Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012).

1.1. Wprowadzenie do języka R

Język R wywodzi się z **języka S**, przeznaczonego do interaktywnej analizy danych, którego początki sięgają 1976 roku. Był on opracowywany przez Johna Chambersa i jego współpracowników (m.in. Ricka Beckera, Trevora Hastiego, Williama Clevelanda i Allana Wilksa) z Bell Laboratories (oddział badawczy i wdrożeniowy firmy telekomunikacyjnej Nokia). Sam **język R** został stworzony na początku lat 90 XX. wieku (choć wersję 1.0 wydano dopiero w 2000 roku) przez Roberta Gentlemana i Rossa Ihakę, którzy pracowali na Wydziale Statystyki Uniwersytetu w Auckland; stanowił początkowo pomoc dydaktyczną do nauki statystyki. Jednakże będąc otwartym projektem, szybko zyskiwał na popularności. Nazwa języka R pochodzi od wspólnej pierwszej litery imienia obu jego autorów, a także świadczy o powiązaniu z językiem S. Obecnie rozwojem R kieruje fundacja *The R Foundation for Statistical Computing* (Biecek, 2017; Gągolewski, 2014; "R (programming language)", 2025; "S (programming language)", 2025; "The R Foundation", 2025).

Najważniejsze cechy języka R (Biecek, 2017; Gągolewski, 2014; Nowosad, 2020; "R (programming language)", 2025; "The R Foundation", 2025) – rys. 1:

- interpretowany język programowania wysokiego poziomu (w języku interpretowanym program nie jest kompilowany, jak np. w językach C lub C++, tylko jest przechowywany w postaci kodu źródłowego; dopiero podczas uruchomienia jest wczytywany, interpretowany i wykonywany przez interpreter

języka),

- wolne i otwarte oprogramowanie – jest darmowy zarówno w zakresie zastosowań edukacyjnych i naukowych, jak i biznesowych (komercyjnych),
- wieloplatformowość – dostępny jest na systemy operacyjne *Windows*, *Linux* i *macOS*,
- przeznaczenie w szczególności do przeprowadzania obliczeń statystycznych i numerycznych, analizy danych i tworzenia grafiki (o wysokiej jakości),
- zwięzła składnia – tzn. „mało kodu powoduje duży efekt”,
- bardzo duża społeczność użytkowników,
- posiada tysiące dodatkowych pakietów mających różne zastosowania, szczególnie w repozytorium **CRAN** (ang. *Comprehensive R Archive Network*, czyli oficjalnym archiwum pakietów R),
- bogata dokumentacja dostępna w postaci dokumentów PDF lub stron HTML,
- biorąc pod uwagę zastosowania w zakresie obliczeń naukowych, przeznaczony jest do wykonywania obliczeń numerycznych,
- możliwość korzystanie z funkcji zaimplementowanych w innych językach, takich jak C oraz C++, a z drugiej strony pozwala także na wywoływanie funkcji języka R z poziomu zewnętrznych środowisk, m.in. w językach Java, C, C++ oraz w narzędziach takich jak Statistica, czy SAS.

Rys. 1. Cechy języka R (grafikę wygenerowano z wykorzystaniem generatywnej sztucznej inteligencji w ramach licencji użytkownika)

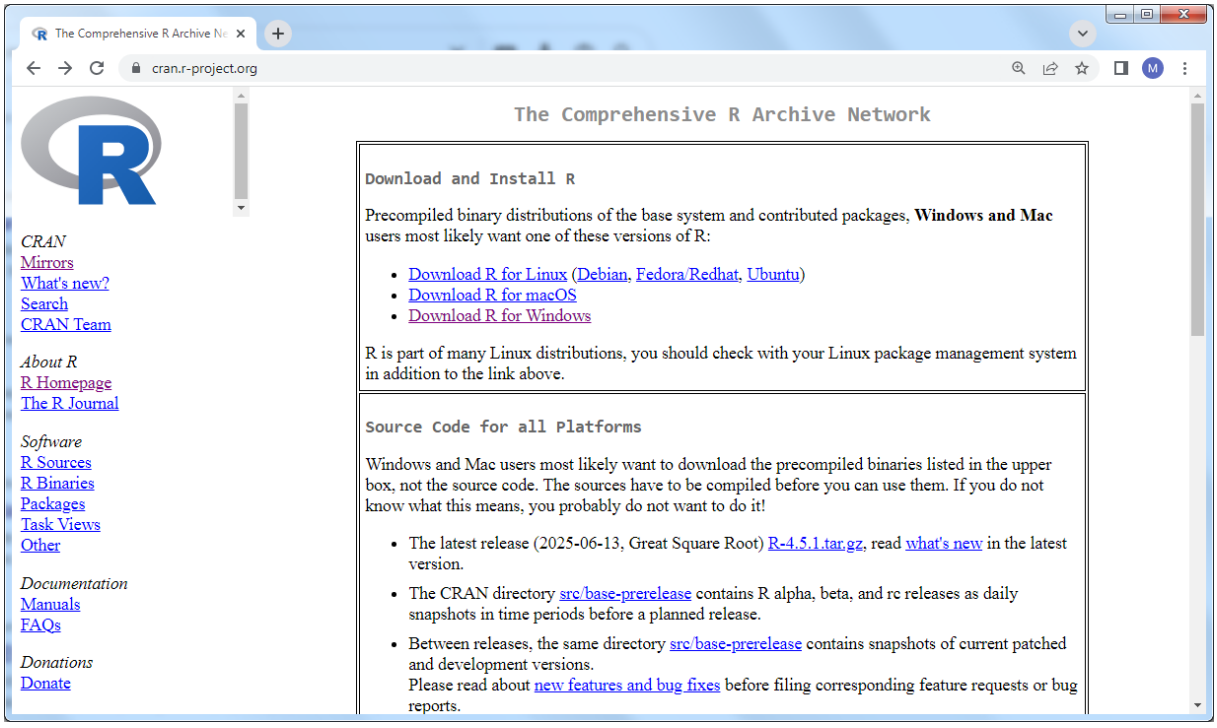
[Tekst alternatywny. Rysunek 1 przedstawia infografikę na temat wybranych cech języka R. Widoczne są ikony symbolizujące poszczególne właściwości: interpretowany język wysokiego poziomu, wolne i otwarte oprogramowanie, wieloplatformowość, przeznaczenie do analizy danych i obliczeń numerycznych, zwięzła składnia, duża społeczność użytkowników, tysiące pakietów, bogata dokumentacja oraz integracja z innymi narzędziami.]

1.2. Przygotowanie środowiska pracy

Przed przystąpieniem do pracy z językiem R należy zainstalować odpowiednie oprogramowanie, które umożliwi interpretację kodu oraz wykonywanie programów napisanych w tym języku. Sam proces instalacji nie jest skomplikowany i nie powinien sprawiać większych problemów.

1.2.1. Podstawowe środowisko R

Po pierwsze, należy zainstalować podstawowe środowisko R (zwane **base** – interpreter i podstawowy zbiór pakietów). Posiada ono duże możliwości w zakresie analizy danych. R, dla każdej platformy, można pobrać ze strony https://cran.r-project.org (“The Comprehensive R Archive Network“, 2025) – rys. 2.

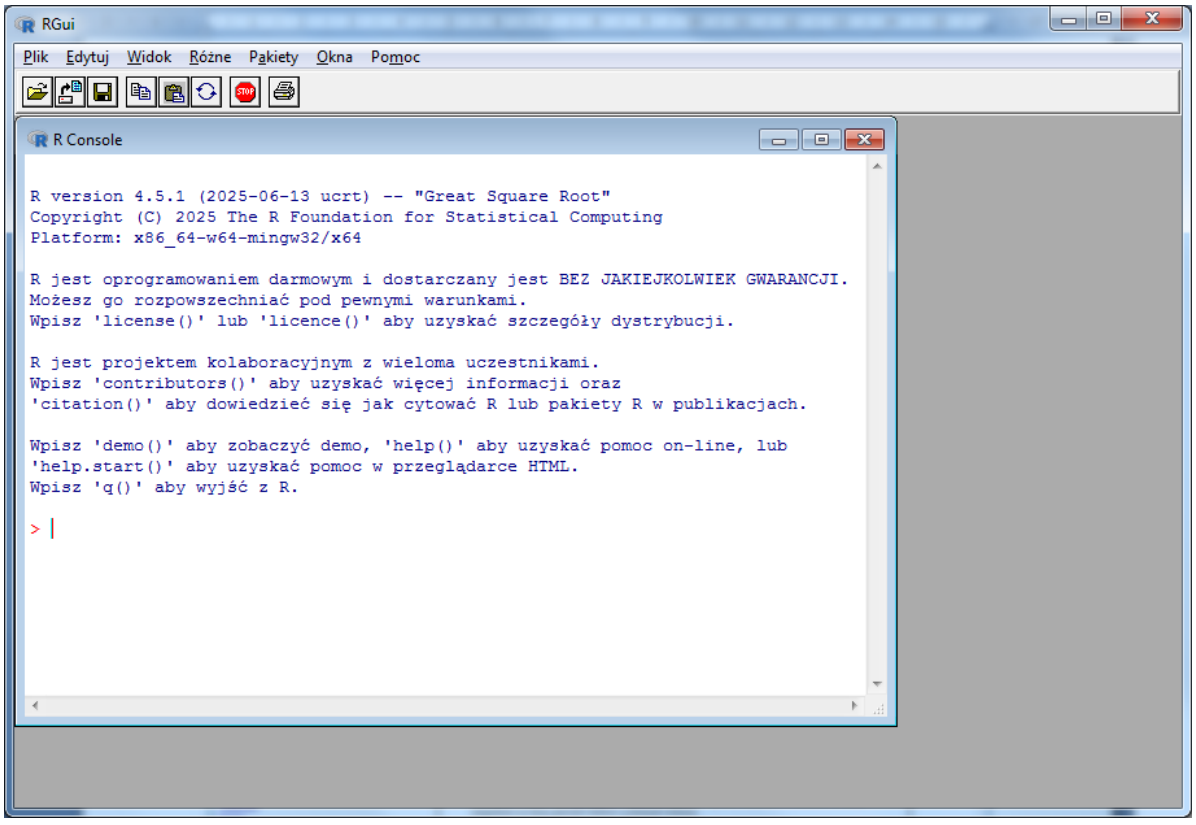


Rys. 2. Pobieranie podstawowego środowiska R – strona https://cran.r-project.org

[Tekst alternatywny. Rysunek 2 przedstawia zrzut ekranu obejmujący stronę internetową https://cran.r-project.org, czyli oficjalne repozytorium projektu R. Na stronie widoczne są różne opcje pobrania instalatora języka R dla różnych systemów operacyjnych (Windows, macOS, Linux) oraz pewne dodatkowe informacje.]

Na stronie głównej (widocznej na rys. 2) dostępne są trzy odnośniki pozwalające na pobranie R dla wybranego systemu operacyjnego; po wybraniu jednej z opcji należy pobrać plik instalacyjny i uruchomić instalator. W momencie tworzenia niniejszych materiałów, aktualna wersja R to **4.5.1**.

Po procesie instalacji można już korzystać z R – rys. 3. Praca z tym programem jest interaktywna. Udostępnia on prosty interfejs wiersza poleceń (**CLI** – ang. *Command Line Interface*). Praca w R jest właśnie często utożsamiana z „konsolą, w której mruga kursor, gdzie można wpisywać polecenia, a następnie odczytywać wyniki" (Biecek, 2017; Lander, 2018).

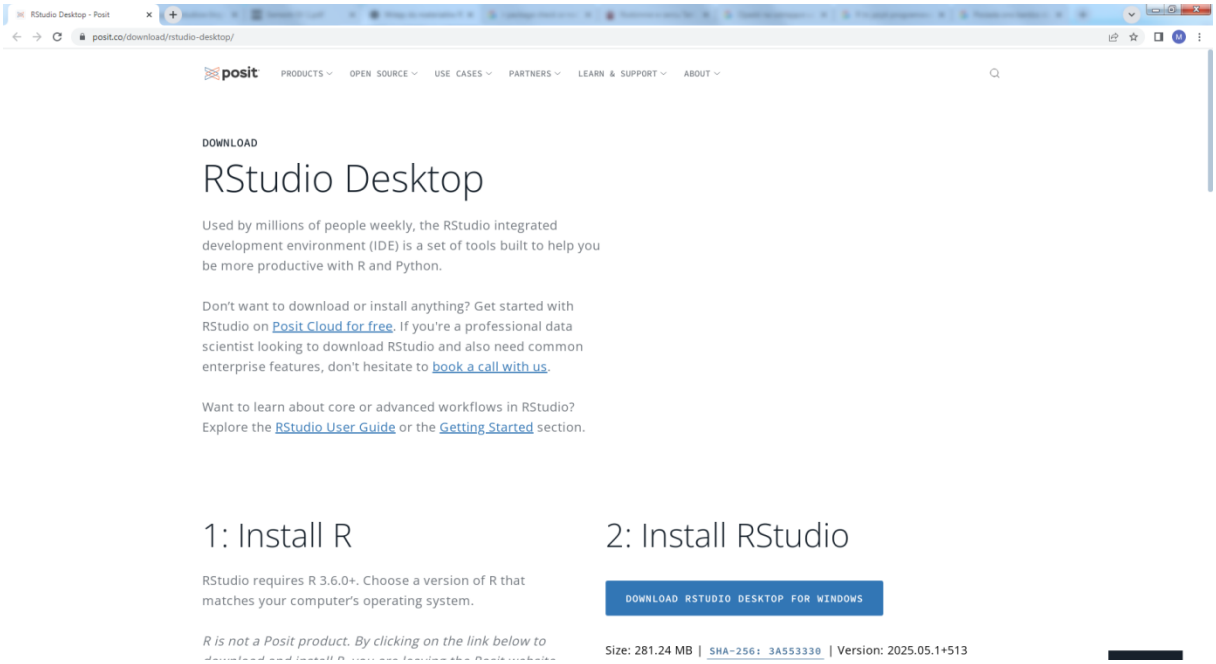


Rys. 3. Podstawowe środowisko R

[Tekst alternatywny. Rysunek 3 przedstawia podstawowe środowisko R w wersji 4.5.1. Widoczne jest okno z tytułem „R Console", w którym użytkownik może wpisywać polecenia w języku R.]

1.2.2. RStudio Desktop

Choć można korzystać z R w podstawowej formie, to praca w takim środowisku bywa niewygodna, zwłaszcza w przypadku bardziej rozbudowanych projektów. Z tego powodu zalecane jest korzystanie z dedykowanego, zintegrowanego środowiska programistycznego (**IDE** – ang. *Integrated Development Environment*). Dla R dostępnych jest kilka różnych IDE, np. RStudio, Rattle, StatET. Obecnie jednym z najpopularniejszych jest **RStudio Desktop** i będzie on wykorzystywany w niniejszym opracowaniu (Biecek, 2017; Gągolewski, 2014; Lander, 2018; Nowosad, 2020). Choć istnieją wersje komercyjne, dostępny jest on także w wersji całkowicie darmowej; za jego rozwój odpowiada Posit Software, PBC. Bezpłatną wersję można pobrać ze strony: https://posit.co/download/rstudio-desktop/ (“RStudio Desktop“, 2025) – rys. 4. Istnieje także wersja typu *Cloud* – umożliwiająca dostęp do narzędzia za pomocą przeglądarki internetowej.

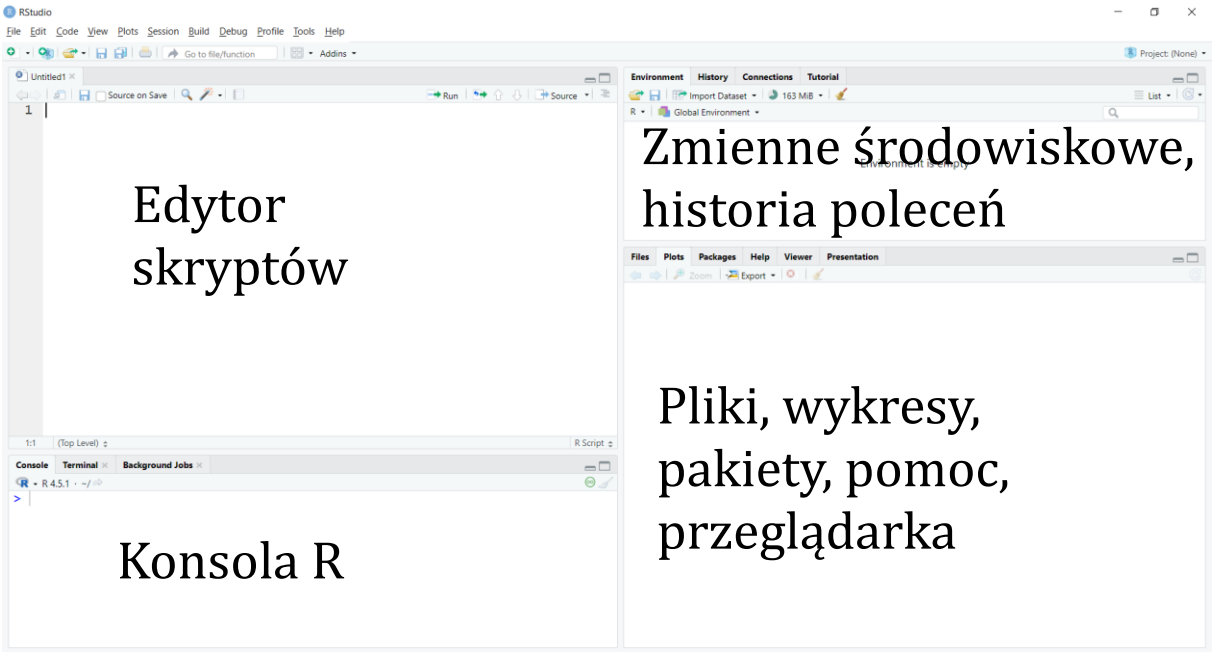


Rys. 4. Pobieranie RStudio Desktop – strona https://posit.co/download/rstudio-desktop

[Tekst alternatywny. Rysunek 4 przedstawia zrzut ekranu obejmujący stronę internetową https://posit.co/download/rstudio-desktop, na której można pobrać program RStudio Desktop dla różnych systemów operacyjnych (Windows, macOS, Linux).]

RStudio Desktop to zintegrowane środowisko programistyczne (IDE) przeznaczone do pracy z językiem R, dostępne dla systemów operacyjnych z rodzin Microsoft Windows, Linux oraz macOS. Zostało ono zaprojektowane z myślą o obliczeniach statystycznych oraz wizualizacji wyników analiz, znacząco ułatwiając codzienną pracę z R. Uwaga: RStudio nie będzie działać bez zainstalowania podstawowej (bazowej) wersji R. Interfejs RStudio (rys. 5) składa się z kilku funkcjonalnych paneli:

- po lewej stronie, na górze, znajduje się edytor kodu źródłowego dla otwartych plików/ skryptów oraz podgląd własności obiektów,
- po prawej stronie, na górze, prezentowana jest m.in. lista aktualnie zadeklarowanych obiektów oraz historia wykonywanych poleceń,
- po lewej stronie, na dole, znajduje się interaktywna konsola R,
- po prawej stronie, na dole, znajdują się m.in. zakładki do wizualizacji wykresów, zarządzania pakietami, plikami oraz dokumentacją, co dopełnia funkcjonalności środowiska (Biecek, 2017; Gągolewski, 2014; Lander, 2018; Nowosad, 2020; “RStudio“, 2025).

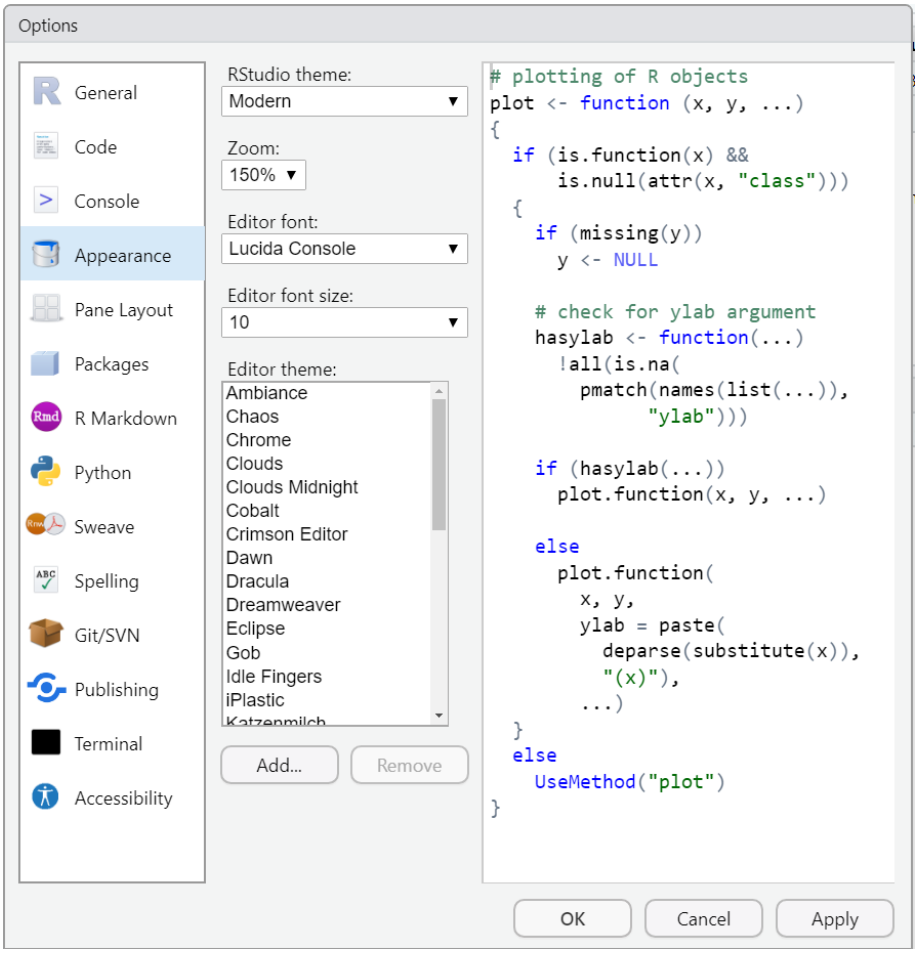


Rys. 5. Interfejs programu RStudio Desktop

[Tekst alternatywny. Rysunek 5 przedstawia Interfejs programu RStudio Desktop. Po lewej stronie, na górze, znajduje się edytor kodu źródłowego. Po prawej stronie, na górze, prezentowana jest m.in. lista aktualnie zadeklarowanych obiektów oraz historia wykonywanych poleceń. Po lewej stronie, na dole, znajduje się interaktywna konsola R. Po prawej stronie, na dole, znajdują się m.in. zakładki do wizualizacji

wykresów, zarządzania pakietami, plikami oraz dokumentacją, co dopełnia funkcjonalności środowiska.]

Aby spersonalizować wygląd programu RStudio Desktop, w menu głównym należy wybrać opcję: ***Tools → Global Options… → Apperance*** – rys. 6.



Rys. 6. Personalizacja wyglądu interfejsu programu RStudio Desktop

[Tekst alternatywny. Rysunek 6 przedstawia możliwość personalizacji wyglądu interfejsu programu RStudio Desktop: Tools → Global Options… → Apperance, gdzie można m.in. wybrać motyw, czy też zdefiniować wielkość i rodzaj wykorzystywanej czcionki.]

1.2.3. Pakiety w R i ich istotność

Na koniec nie tylko warto, a nawet trzeba, doinstalować dodatkowe pakiety z różnymi, przydatnymi w danej chwili funkcjami. O sile R świadczy właśnie

możliwość instalacji pakietów. Samo podstawowe środowisko R można porównać do „smartfona przywróconego do ustawień fabrycznych" – który jest bardzo przydatny i można w nim zrobić wiele różnych rzeczy. Jednak w pewnej chwili może zabraknąć pewnej aplikacji (czyli właśnie pakietu). W takiej sytuacji należy wykonać odpowiednik instalacji aplikacji – czyli instalację pakietu.

Pakiet można zdefiniować jako swoistą „aplikacje języka R" – tj. fragmenty kodu zawierające funkcje, zbiory danych, dokumentację i nie tylko. Pakiety rozbudowują R o nowe funkcjonalności. Można je znaleźć we wspomnianym wcześniej **repozytorium CRAN**, czyli „sklepie z aplikacjami dla języka R". Należy zaznaczyć jednak, że możliwe jest również pobieranie pakietów z innych miejsc.

Najważniejsze informacje o pakietach w R:

- pakiety instaluje się za pomocą polecenia: **install.packages('*nazwa_pakietu*')**. W trakcie instalacji pakietu może się zdarzyć, iż wystąpi jakiś błąd, co może to być spowodowane tym, że pakiet, który próbuje się zainstalować ma tak zwaną „zależność". W takiej sytuacji, aby pakiet ten można było poprawnie zainstalować i uruchomić, wymagane jest zainstalowanie innego pakietu(ów).Wtedy warto wykorzystać nieco bardziej rozbudowane polecenie instalujące pakiety, które dodatkowo, automatycznie, doinstaluje również inne, wymagane pakiety (zależności): **install.packages(*'nazwa_pakietu'*, *dependencies = TRUE*)**,
- pakiet instaluje się tylko raz, jednakże należy załadować go podczas każdej sesji za pomocą polecenia: **library(*'nazwa_pakietu'*)**,
- pakietów jest bardzo dużo. Zestawienie *CRAN Task Views* dostępne pod adresem https://cran.r-project.org/web/views/ zawiera informacje o dostępnych pakietach, pogrupowane tematycznie,
- pakiety można również zaktualizować z poziomu menu RStudio Desktop: ***Tools → Install Packages…*** ,
- do usunięcia pakietu służy polecenie: **remove.packages(*'nazwa_pakietu'*)**,
- w celu zaktualizowania pakietów z repozytorium CRAN, należy wpisać do konsoli polecenie: **update.packages()**. Alternatywnie, można je zaktualizować z poziomu menu RStudio Destktop: ***Tools → Check for Package Updates***.

W codziennej pracy z R przydatne może być polecenie, które przy wykorzystaniu instrukcji warunkowej (por. podrozdział 4.1.) sprawdza, czy dany ***pakiet*** jest zainstalowany: jeśli tak, jest on wczytywany, a jeśli nie, najpierw jest instalowany (wraz z zależnościami), a następnie dopiero wczytywany:

```
if (!require(pakiet)) {
        install.packages("pakiet", dependencies = TRUE)
        library(pakiet)
} else {library(pakiet)}
```

Należy pamiętać że pakiety, R oraz RStudio są stale ulepszane. Warto więc od czasu do czasu sprawdzać dostępność aktualizacji. Przykładowo, w celu aktualizacji RStudio, należy wybrać polecenie ***Help → Check for Updates*** z menu górnego.

1.3. Podstawy pracy z R

Aby wywołać polecenie(a) w języku R, należy wpisać je w konsoli obok znaku zachęty **>** i nacisnąć klawisz ***Enter***. Aby powtórzyć wiersz kodu, należy nacisnąć przycisk ***Strzałka w górę***. W systemie Windows można przerwać wykonywanie kodu za pomocą klawisza ***Esc***. Jeśli znak zachęty **>** zmienił się na znak **+** oznacza to, że R oczekuje na dokończenie wprowadzania wyrażenia w kolejnym wierszu.

R może pracować w dwóch trybach:

- **interaktywnym** – gdzie po każdym wydanym poleceniu należy nacisnąć klawisz ***Enter*** , po czym natychmiast otrzymywany jest wynik działania tego polecenia w konsoli,
- **wsadowym** (ang. *batch mode*) – w którym zleca się środowisku R uruchomienie określonego pliku źródłowego (nazywanego **skryptem**) – tj. pliku tekstowego, najczęściej o rozszerzeniu **.R**, zawierającego kolejne polecenia języka R przeznaczone do wykonania.

Kilka podstawowych aspektów dotyczących pracy z R:

- R rozróżnia wielkość liter,
- rolę separatora części dziesiętnej liczby pełni kropka (a nie przecinek),
- przy zamykaniu R nie jest zapisywana zawartość pamięci,
- jednym z podstawowych rodzajów danych w programie R są **wektory**. Mogą one zawierać liczby, napisy, wartości logiczne lub inne typy. **W języku R nawet pojedyncza wartość jest wektorem, tyle że jednoelementowym**,
- **w języku R wszystko jest obiektem**.

1.3.1. Komentarze

Komentarze pomagają innym użytkownikom (oraz samym autorom skryptów) opisać, zrozumieć i pozwalać zapamiętać, do czego służy dany fragment kodu. Komentarz tworzy się za pomocą znaku kratki: #.

Język R nie wykonuje kodu objętego komentarzem – tj. od pierwszego wystąpienia symbolu #, aż do końca danego wiersza. W R nie ma komentarzy blokowych; przykładowo, w skrypcie, do tworzenia i usuwania komentarza, wykorzystywany jest skrót klawiszowy ***Ctrl*** + ***Shift*** + ***c*** – zostaje wstawiony automatycznie komentarz w każdej linii z zaznaczonego bloku kodu.

1.3.2. Skrypty

Chociaż do wykonywania najprostszych zadań nie ma potrzeby pisania programów (gdyż można pracować w trybie interaktywnym), to jednak bardzo często zachodzi potrzeba napisania własnego **skryptu** (tj. zbioru instrukcji, poleceń programu R). Plik skryptowy programu R to zwykły plik tekstowy z rozszerzeniem .R. Nowy skrypt w RStudio Desktop (rys. 7) można utworzyć w dwojaki sposób:

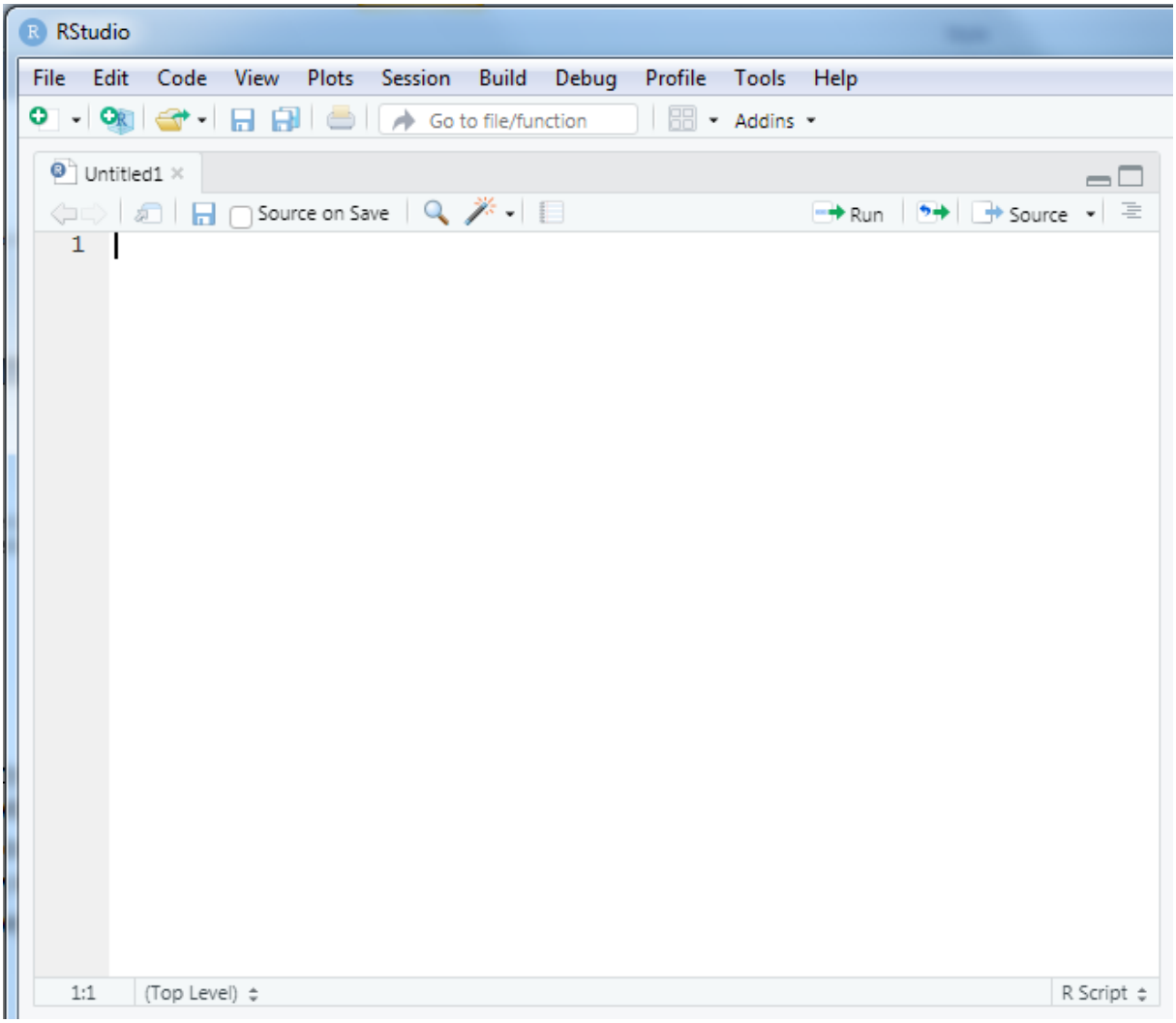
a) wybrać z menu górnego ***File → New File → R Script***,
b) nacisnąć kombinację klawiszy: ***CTRL*** + ***SHIFT*** + ***n***.

Skrypt warto od razu zapisać na dysku twardym w wybranej lokalizacji: ***File → Save As***; można także użyć kombinacji klawiszy: ***CTRL*** + ***s***.

Ładowanie niezbędnych pakietów powinno nastąpić na początku skryptu. Warto wyrobić sobie nawyk częstego zapisywania edytowanych plików.

R robi tylko i wyłącznie te rzeczy, o których zrobienie „zostanie wprost poproszony” – by wydrukować (wyprowadzić) jakiś tekst, należy wykorzystać więc odpowiednią funkcję, np. **print()** lub **cat()**.

[Tekst alternatywny. Rysunek 7 przedstawia fragment okna programu RStudio Desktop – nowy, pusty skrypt o nazwie Untitled1.]



Rys. 7. Nowy skrypt w programie RStudio Desktop

Wybrane, przydatne skróty klawiszowe podczas pracy ze skryptami w RStudio Desktop:

a) ***CTRL*** + ***Shift*** + ***Enter*** – wykonuje całość kodu ze skryptu,
b) ***Ctrl*** + ***Enter*** – wykonuje wybrane (a więc kilka zaznaczonych linii kodu) lub jedną linię kodu – tą, w której jest umieszczony kursor,
c) ***Shift*** + ***Strzałka w górę*** (lub ***Shift*** + ***Strzałka w dół***) – zaznaczanie fragmentu kodu bez wykorzystania myszy,
d) ***Ctrl*** + ***Home*** – powrót na początek skryptu,
e) ***Ctrl*** + ***End*** – przejście do końca skryptu,
f) ***Ctrl*** + ***1*** oraz ***Ctrl*** + ***2*** – przenoszenie kursora pomiędzy edytorem skryptów, a konsolą,
g) ***TAB*** *lub* ***Ctrl*** + ***Spacja*** – uruchamia mechanizm podpowiedzi i uzupełniania nazw obiektów.

1.3.3. Przechowywanie danych przez RStudio. Katalog roboczy

W celu sprawdzenia, w jakim katalogu zainstalowany jest R, można wykorzystać funkcję **R.home().**

Aby sprawdzić adres katalogu, w którym bieżąca sesja R przechowuje dane tymczasowe (ang. *temporary directory*), należy wykorzystać funkcję **tempdir()**. Uwaga: Po zamknięciu R pliki te nie będą dalej dostępne, a lokalizacja tego katalogu może ulec zmianie. Aby sprawdzić, w jakim katalogu zainstalowany jest R, należy wykorzystać funkcję **R.home().** Do „okienkowego" utworzenia ścieżki katalogu do pliku przydatna jest funkcja **file.choose()** – po jej wywołaniu wyświetlane jest okno dialogowe, w którym można wskazać plik znajdujący się na dysku.

Istotnym pojęciem jest **katalog roboczy** (bieżący) użytkownika (ang. *working directory*) – jest to folder na dysku, w którym R domyślnie szuka plików do wczytania oraz zapisuje wyniki pracy (np. dane, wykresy, pliki wynikowe). Można go więc traktować jako swoisty „punkt wyjścia" dla wszystkich operacji na plikach wykonywanych w sesji R. W ogólności (por. rozdział 6), ścieżki bezwzględne identyfikują jednoznacznie położenie pliku lub katalogu na dysku, niezależnie od kontekstu. Istnieje jednakże możliwość operowania na ścieżkach względnych, odnoszących się do bieżącego katalogu roboczego. Ustawienie właściwego folderu roboczego jest istotne, gdyż pozwala właśnie na korzystanie ze ścieżki względnej; umożliwia to określanie ścieżki pliku w odniesieniu do katalogu roboczego. Do sprawdzenia, który katalog jest katalogiem bieżącym, służy polecenie **getwd()**, a do zmiany katalogu bieżącego – funkcja **setwd()**. W RStudio Desktop katalog roboczy można zmienić wygodnie w formie „okienkowej" za pomocą kombinacji klawiszy ***Ctrl*** + ***Shift*** + ***H***.

Podczas pracy z R katalog roboczy jest więc bardzo istotny, gdyż ułatwia organizację pracy. Dzięki temu można trzymać wszystkie pliki danego projektu w jednym miejscu. Wczytywanie i zapisywanie danych jest możliwe bez konieczności podawania pełnych ścieżek do plików.

Zadanie 1.1.

Korzystając z dowolnych narzędzi (np. wyszukiwarki internetowej), znajdź wiarygodne informacje na temat rzeczywistych zastosowań języka R. Przez jakie

duże przedsiębiorstwa jest on wykorzystywany? Czy w szeroko pojętej analizie danych lepiej wykorzystywać język R, czy język Python?

Podpowiedź:
Na obecną chwilę zarówno język R, jak i język Python stanowią bardzo silne narzędzia w zakresie analizy danych, a także uczenia maszynowego. Mają odmienne mocne i słabe strony.

1.3.4. Komunikaty

Oprócz wyznaczenia właściwego wyniku wykonywanej operacji, R może także generować różnego rodzaju komunikaty:

- **błędy** (ang. *errors*) – oznaczają sytuację, ze działanie danej funkcji musi zostać natychmiast przerwane, gdyż nie może być poprawnie kontynuowane, np. obliczenie logarytmu z napisu **ID**,
- **ostrzeżenia** (ang. *warnings*) – informują o potencjalnym problemie podczas wykonywania funkcji. Pomimo wystąpienia trudności obliczenia są kontynuowane, lecz wynik może wymagać dodatkowej weryfikacji przez programistę – musi on przejrzeć się dokładnie wykonywanej funkcji, np. obliczenie logarytmu z wartości ujemnej zwraca wartość **NaN**,
- **wiadomości** (ang. *messages*) – mają charakter czysto informacyjny; są to dodatkowe informacje pojawiające się w trakcie działania funkcji.

1.3.5. System pomocy – uzyskiwanie informacji o funkcjach i pakietach

Język R wyposażony jest w rozbudowany system pomocy, który pozwala na szybkie uzyskiwanie informacji o funkcjach, pakietach, a także dostępnych opcjach. Dokumentacja ta tworzona jest przez autorów pakietów, przez co stanowi więc główne źródło wiedzy podczas pracy z językiem R.

Podstawową metodą uzyskiwania informacji (pomocy) o funkcji jest użycie znaku zapytania przed jej nazwą (**?**) – np. **?plot**, lecz alternatywnie można wykorzystać funkcję **help()** – np. **help(*"print"*)**. Skorzystać można także z polecenia **help.search(*"fraza_kluczowa"*)** – w wyniku pojawia się lista funkcji oraz pakietów, w których występuje wprowadzona fraza kluczowa, np. **help.search(*"x-y plot"*)**.

Pomoc w R obejmuje także dokumentację całych pakietów. Po zainstalowaniu i wczytaniu pakietu, można uzyskać listę dostępnych funkcji i przykładów za pomocą

polecenia **help(*package* = *"nazwa_pakietu"*)**, np. **help(*package* = *"dplyr"*)**. Dodatkowo, wiele pakietów zawiera tzw. *vignettes*, czyli rozbudowane samouczki prezentujące typowe zastosowania. Dostęp do nich można uzyskać przy wykorzystaniu funkcji **vignette(*package* = *"nazwa_pakietu"*)** – np. **vignette(*package* = *"dplyr"*)** lub krócej: **vignette(*"dplyr"*)**. Warto także wspomnieć o poleceniu **example()**, które pozwala uruchomić przykłady zawarte w dokumentacji (o ile istnieją).

W RStudio Desktop przeglądanie dokumentacji możliwe jest w dedykowanej zakładce ***Help***, dzięki czemu Użytkownik na każdym etapie pracy z R ma bieżący dostęp do informacji dotyczących funkcji i pakietów, bez konieczności korzystania z zewnętrznych źródeł.

Oprócz dokumentacji wbudowanej w środowisko R, cennym uzupełnieniem są źródła zewnętrzne. Oficjalne repozytorium **CRAN** (“The Comprehensive R Archive Network“, 2025) udostępnia dokumentację każdego pakietu wraz z opisem funkcji i przykładami. Serwis **Rdocumentation** (“Rdocumentation“, 2025) pozwala na szybkie przeszukiwanie dokumentacji online, natomiast platformy społecznościowe (takie jak np. **Stack Overflow**) umożliwiają wymianę doświadczeń z innymi użytkownikami.

Zadanie 1.2.

Korzystając z dowolnej przeglądarki internetowej zapoznaj się z oficjalną dokumentacją języka R na oficjalnej stronie repozytorium CRAN (https://cran.r-project.org/) w zakładce ***Documentation → Manuals***.

2. R jako rozbudowany kalkulatoR

Język R stanowi potężne narzędzie, które pozwala na wykonywanie różnorodnych obliczeń, czy też manipulowanie danymi. Niezależnie od tego, może zostać także wykorzystany do wykonywania podstawowych działań matematycznych. Język R posiada specyficzne dla siebie możliwości wykonywania działań matematycznych, zmienne, funkcje i typy danych.

Przy tworzeniu niniejszego rozdziału korzystano z pomocy programu R, dokumentacji R ("Rdocumentation", 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN ("The Comprehensive R Archive Network", 2025) i jego podstron (gdzie znajdują się m.in. opisy poszczególnych pakietów i funkcji) oraz pozycji literaturowych (Biecek, 2018; Gągolewski, 2014; Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012).

2.1. Podstawowe operatory i funkcje matematyczne

W języku R stosowana jest standardowa kolejność wykonywania działań: działania w nawiasach, potęgowanie i pierwiastkowanie, mnożenie i dzielenie, dodawanie i odejmowanie. Wynika z tego, iż operacje umieszczanie w nawiasach są wykonywane przed innymi działaniami (tj. mają wyższy priorytet). W R potęgowanie jest prawostronnie łączone, np. **2^3^2** to **512**, gdyż liczone jest jako **2^(3^2)**, a nie **(2^3)^2**.

Podstawowe operatory arytmetyczne zostały przedstawione w tabeli 1, a funkcje matematyczne (oraz stałą π) w tabeli 2.

Tabela 1. Podstawowe operatory arytmetyczne

Operator	Opis
+	Dodawanie
-	Odejmowanie
*****	Mnożenie
/	Dzielenie
^ lub ******	Potęgowanie
%%	Reszta z dzielenia (modulo)
%/%	Część całkowita z dzielenia

Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki" nr FERS.01.05-IP.08-0234/23

Tabela 2. Wybrane funkcje i stałe matematyczne

Funkcja	Opis
abs(*x*)	Wartość bezwzględna z ***x***
log(*x*)	Logarytm naturalny z ***x***
log(*x*, *y*)	Logarytm o podstawie ***y*** z ***x***
log10(*x*)	Logarytm dziesiętny (o podstawie 10) z ***x***
log2(*x*)	Logarytm o podstawie 2 z ***x***
exp(*x*)	Funkcja wykładnicza (eksponenta) z ***x***
sqrt(*x*)	Pierwiastek kwadratowy z ***x*** (równoważne: ***x*^0.5**) Uwaga: Pierwiastki wyższych stopni można wyznaczyć za pomocą operatora potęgowania.
ceiling(*x*)	Zaokrąglenie w górę liczby ***x*** (tzw. „sufit" – do najbliższej największej liczby całkowitej)
floor(*x*)	Zaokrąglenie w dół liczby ***x*** (tzw. „podłoga" – do najbliższej najmniejszej liczby całkowitej)
round(*x*,*y*)	Zaokrąglenie liczby ***x*** do ***y*** miejsc po przecinku
trunc(*x*)	Zwraca wartość ***x*** (będąca liczbą całkowitą) po odcięciu części rzeczywistej
signif(*x*,*k*)	Wartość ***x*** zaokrąglona do ***k*** miejsc znaczących.
combn(*n*,*k*)	Lista wszystkich kombinacji ***k*** elementowych ze zbioru ***n*** elementowego
choose(n,k)	Liczba kombinacji ***k*** elementowych ze zbioru ***n*** elementowego
factorial(*x*)	Silnia z ***x***
pi	Stała π (tj. 3.141593…)

Przykład 2.1.

Podstawowe operacje matematyczne.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***. Rezultaty działania poniższych poleceń przedstawiono na rysunku 8.

```
5+1,11          #dodawanie; błąd!
5+1.11          #dodawanie; rolę separatora dziesiętnego pełni kropka
3-8             #odejmowanie
5*5             #mnożenie
7/3             #dzielenie
7%%3            #reszta z dzielenia (modulo)
7%/%3           #część całkowita z dzielenia
2^4             #potęgowanie (sposób 1)
2**4            #potęgowanie (sposób 2)
abs(-10)        #wartość bezwzględna z -10
```

```
Abs(-10)            #błąd; wielkość liter ma znaczenie
log10(100)          #logarytm dziesiętny ze 100
9^(1/3)             #pierwiastek 3 stopnia z 9
9^1/3               #uwaga na kolejność wykonywania działań!
3*pi+2.1
ceiling(11.9352)
floor(11.9352)
round(11.9352, 1)
round(11.9352, 2)
round(11.9352, 3)
trunc(11.9352, 3)
signif(11.9352, 3)
signif(11.9352, 4)
```

```
> 5+1,11              #dodawanie; błąd!
Error: unexpected ',' in "5+1,"
> 5+1.11              #dodawanie; rolę separatora dziesiętnego pełni kropka
[1] 6.11
> 3-8                 #odejmowanie
[1] -5
> 5*5                 #mnożenie
[1] 25
> 7/3                 #dzielenie
[1] 2.333333
> 7%%3                #reszta z dzielenia (modulo)
[1] 1
> 7%/%3               #część całkowita z dzielenia
[1] 2
> 2^4                 #potęgowanie (sposób 1)
[1] 16
> 2**4                #potęgowanie (sposób 2)
[1] 16
> abs(-10)            #wartość bezwzględna z -10
[1] 10
> Abs(-10)            #błąd; wielkość liter ma znaczenie
Error in Abs(-10) : could not find function "Abs"
> log10(100)          #logarytm dziesiętny ze 100
[1] 2
> 9^(1/3)             #pierwiastek 3 stopnia z 9
[1] 2.080084
> 9^1/3               #uwaga na kolejność wykonywania działań!
[1] 3
> 3*pi+2.1
[1] 11.52478
> ceiling(11.9352)
[1] 12
> floor(11.9352)
[1] 11
> round(11.9352, 1)
[1] 11.9
> round(11.9352, 2)
[1] 11.94
> round(11.9352, 3)
[1] 11.935
> trunc(11.9352, 3)
[1] 11
> signif(11.9352, 3)
[1] 11.9
> signif(11.9352, 4)
[1] 11.94
```

Rys. 8. Rezultat działania poleceń z przykładu 2.1.

*Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki”
nr FERS.01.05-IP.08-0234/23*

[Tekst alternatywny. Rysunek 8 przedstawia fragment okna programu RStudio Desktop – rezultat działania poleceń z przykładu 2.1. w konsoli R.]

Najważniejsze funkcje trygonometryczne zamieszczono w tabeli 3. Uwaga: miara kąta musi być podany w radianach.

Tabela 3. Podstawowe funkcje trygonometryczne

Operator	Opis
cos(*x*)	Wartość funkcji cosinus dla argumentu ***x***
sin(*x*)	Wartość funkcji sinus dla argumentu ***x***
tan(x)	Wartość funkcji tangens dla argumentu ***x***
acos(*x*)	Wartość funkcji arcus cosinus dla argumentu ***x***
asin(*x*)	Wartość funkcji arcus sinus dla argumentu ***x***
atan(*x*)	Wartość funkcji arcus tangens dla argumentu ***x***

W podstawowej wersji R brak jest jednakże funkcji do konwersji stopni na radiany. Wiedząc, że $1° = \frac{\pi}{180}$ można przeliczać wartości ręcznie stosując zależność (1):

$$x\,[rad] = x\,[°] * (\frac{\pi}{180}) \quad (1)$$

gdzie: ***x*** – miara kąta w stopniach; ***π*** – stała matematyczna = 3,141593…

Można także zdefiniować własne funkcje (por. podrozdział 5.1.), np.:
radNAdstopnie <- function(rad) {(rad * 180) / (pi)}
stopnieNArad <- function(stopnie) {(stopnie * pi) / (180)}
Alternatywą jest instalacja dodatkowego pakietu, np. **NISTunits**.

2.2. Zmienne w R

Definicja zmiennej została wprowadzona na wcześniejszych przedmiotach w toku studiów – w razie konieczności należy sięgnąć do dostępnych materiałów.

Zmienne w języku R posiadają unikalną nazwę, przez którą można się do nich odwoływać (np. w skrypcie). Służą do przechowywania danych i wyników wykonywanych operacji. Nazwy obiektów (w tym zmiennych) w R muszą spełniać poniższe kryteria:

- mogą zawierać dowolną kombinację znaków alfanumerycznych, kropek (**.**) i podkreśleń (_),

- nie mogą zaczynać się od cyfry ani znaku podkreślenia – muszą rozpoczynać się od litery,
- rozróżniana jest wielkość liter – przykładowo zmienne o nazwach: **ZMIENNA**, **zmienna**, **Zmienna** to zupełnie trzy różne zmienne,
- zmiennym powinno nadawać się czytelne nazwy, co jest uważane za dobrą praktykę,
- nazwy rozpoczynające się od kropki są wykorzystywane do oznaczania wewnętrznych zmiennych R; nie jest zalecane ich wykorzystywane do innych celów.

O zmiennych w R należy także wiedzieć, że:

- R nie wymaga deklarowania obiektów i określania ich typów przed pierwszym użyciem (język z typowaniem dynamicznym). W ogólności, w R nie istnieje polecenie do deklarowania zmiennej,
- zmienna jest tworzona w momencie pierwszego przypisania jej wartości,
- wartość zmiennej w trakcie działania programu może być wielokrotnie odczytywana lub zastępowana inną,
- aby usunąć zmienną, należy wykorzystać polecenie: **remove(*nazwa_zmiennej*)** lub skróconą wersję: **rm(*nazwa_zmiennej*)**; można usunąć także kilka wskazanych zmiennych jednocześnie: **rm(*list = c("zmienna1", "zmienna2")*)**,
- można usunąć wszystkie zmienne przechowywane w pamięci poleceniem: **rm(*list=ls()*)**,
- wartość do zmiennej przypisuje się z użyciem jednego z poniższych **operatorów przypisania** (**podstawienia**); co prawda można je stosować zamiennie, aby zachować większą czytelność kodu, należy zdecydować się na jeden z wariantów:
  - **=**
  - **<-** (na zajęciach wykorzystywany będzie ten operator),
- zmienna **.Last.value** przechowuje wynik ostatnio wykonanego polecenia.

W RStudio Desktop, w oknie po prawej stronie na górze (zakładka **Environment**), wyświetlane są zmienne i ich wartości – dzięki temu można w każdej chwili sprawdzić, jakie zmienne aktualnie są pamiętane, a także jakie są wartości tych zmiennych.

Podsumowując, w R rezultaty działania poleceń mogą być przypisywane do zmiennych. Dzięki temu, zamiast być wyświetlane od razu na ekranie, zostaną one zapamiętane w pamięci programu R i będzie można wykorzystywać je w przyszłości.

Przykład 2.2.

Oblicz wartości funkcji trygonometrycznych: sinus, cosinus i tangens dla kąta α=50°.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***. Rezultaty działania poniższych poleceń przedstawiono na rysunku 8.

```
alpha_deg <- 50
alpha_rad <- alpha_deg * pi / 180  #zamiana stopni na radiany

sin(alpha_rad)   #sinus
cos(alpha_rad)   #cosinus
tan(alpha_rad)   #tangens
```

2.3. Pobieranie danych od użytkownika

W wielu sytuacjach podczas pracy z R niezbędne jest dynamiczne wczytanie danych – np. wprowadzonych przez użytkownika w konsoli podczas działania programu. Takie podejście zwiększa elastyczność i interaktywność skryptów, umożliwiając ich łatwe dostosowanie do różnych przypadków bez konieczności edycji kodu źródłowego; dane nie są zapisane „na sztywno” w kodzie źródłowym.

Do pobierania danych wejściowych od użytkownika w konsoli R można wykorzystać m.in. jedną z dwóch funkcji:

a) **scan()** – w ogólności funkcja ta pozwala na wczytanie danych do wektora lub listy z konsoli lub pliku. Aby wczytać dane wejściowe konkretnego typu (**logical**, **integer**, **numeric**, **complex**, **character**, **raw**, **list**), należy określić typ wprowadzanej wartości, do czego służy argument o nazwie ***what***, np.: **zmienna <- scan(*what = numeric()*)**,

b) **readline()** – służy do czytania linii z terminala w trybie interaktywnym i zwraca jednoelementowy wektor znaków. Aby pobrać wartość innego typu, należy wykonać odpowiednią konwersję, np. przy wykorzystaniu funkcji konwersji **as.numeric()** dla typu numerycznego: **as.numeric(readline(*prompt="Informacja dla użytkownika"*))**.

Przykład 2.2.

Zmienne. Pobieranie danych od użytkownika.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***. Rezultaty działania poniższych poleceń przedstawiono na rysunku 9.

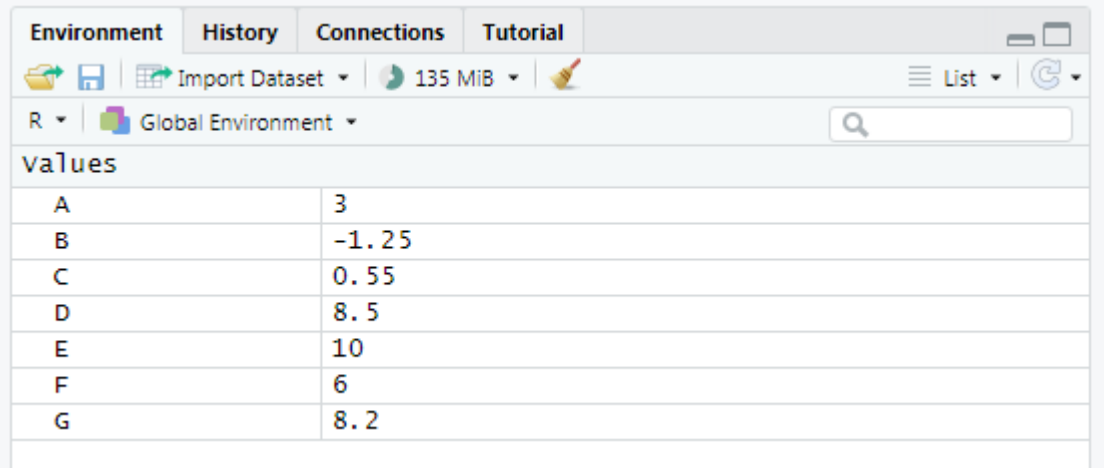
```
A<-3          #przypisanie wartości do zmiennej A, sposób nr 1
print(A)      #print() to funkcja, która drukuje tekst; tak jak i cat()
print(a)      #błąd, dlaczego?
B = -1.25     #przypisanie wartości do zmiennej B, sposób nr 2
print(B)
C <- 0.55     #wykorzystanie spacji umożliwia pisanie czytelniejszego kodu
D <- scan(what = numeric())    #pobranie wartości numerycznej w konsoli
E <- as.numeric(readline(prompt="E = "))  #jw., sposób nr 2
F <- factorial(A)       #silnia
G <- B - C + E
Print(F)                #dlaczego tu jest błąd?
print(F)
cat(G)
H <- 3A + B             #dlaczego tu jest błąd?
```

```
> A<-3            #przypisanie wartości do zmiennej A, sposób nr 1
> print(A)        #print() to funkcja, która drukuje tekst; tak jak i cat()
[1] 3
> print(a)        #błąd, dlaczego?
Error: object 'a' not found
> B=-1.25         #przypisanie wartości do zmiennej B, sposób nr 2
> print(B)
[1] -1.25
> C <- 0.55     #wykorzystanie spacji umożliwia pisanie czytelniejszego kodu
> D <- scan(what = numeric())    #pobranie wartości numerycznej w konsoli
1: 8.5
2:
Read 1 item
> E <- as.numeric(readline(prompt="E = "))  #jw., sposób nr 2
E = 10
> F <- factorial(A)
> G <- B - C + E
> Print(F)      #dlaczego tu jest błąd?
Error in Print(F) : could not find function "Print"
> print(F)
[1] 6
> cat(G)
8.2
> H <- 3A + B   #dlaczego tu jest błąd?
Error: unexpected symbol in "H <- 3A"
```

Rys. 9. Rezultat działania poleceń z przykładu 2.2.

[Tekst alternatywny. Rysunek 9 przedstawia fragment okna programu RStudio Desktop – rezultat działania poleceń z przykładu 2.2. w konsoli R.]

Po prawej stronie programu RStudio Desktop można sprawdzić, jak kształtują się aktualne wartości przechowywanych zmiennych – rys. 10.



Rys. 10. Aktualne wartości zmiennych – po wykonaniu kodu z rys. 9.

[Tekst alternatywny. Rysunek 10 przedstawia fragment okna programu RStudio Desktop – aktualne wartości zmiennych po wykonaniu kodu z rys. 9, np. A = 3.]

Przykład 2.3.

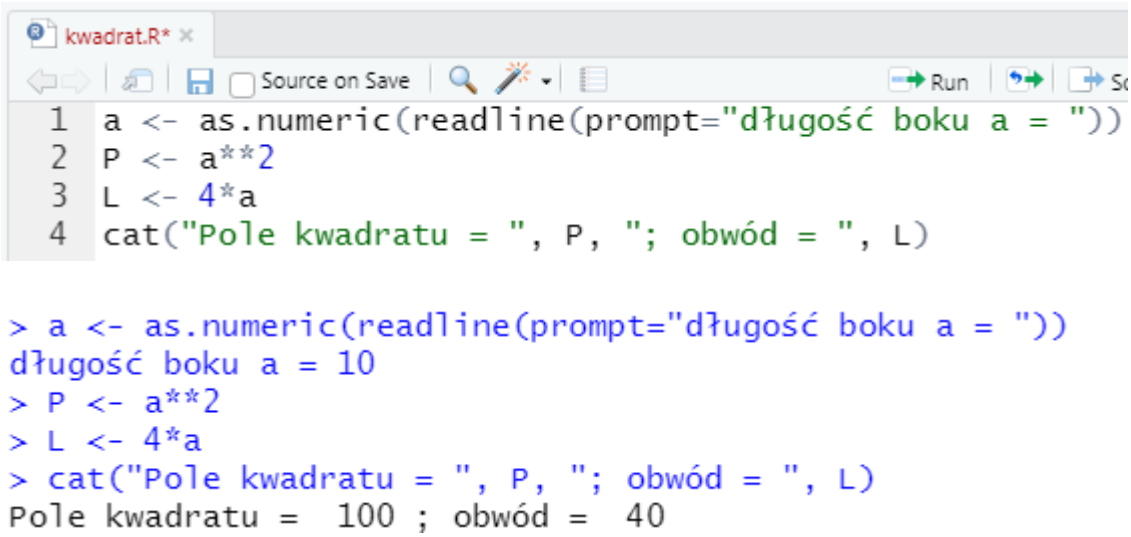
Napisz program, który ma obliczyć pole i obwód kwadratu o długościach boku a (wartość ta ma być pobierana od użytkownika w konsoli), a następnie wyprowadzi (wydrukuje) wyniki na ekran.

W tym celu utwórz nowy skrypt w programie RStudio Desktop o nazwie **kwadrat.R**, Uwaga: kod należy uruchamiać linijka po linijce! Przykładowe rozwiązanie przedstawiono na rysunku 11.

Sprawdź co się stanie, gdy nie zostanie podana poprawna wartość liczbowa – np. jakiś tekst. Zastanów się jak temu zaradzić i przechwycić potencjalne błędy?

Podpowiedź:

Przypomnienie: aby uruchomić dany fragment kodu, tzn. „przenieść pewną linię kodu z okna ze skryptem do terminala", należy posłużyć się skrótem ***Ctrl*** + ***Enter*** – wówczas aktywna linia kodu zostanie wykonana. Jeśli chcemy jednorazowo uruchomić więcej niż jedną linię kodu to można zaznaczyć je wszystkie, a następnie posłużyć się skrótem ***Ctrl*** + ***Enter***.

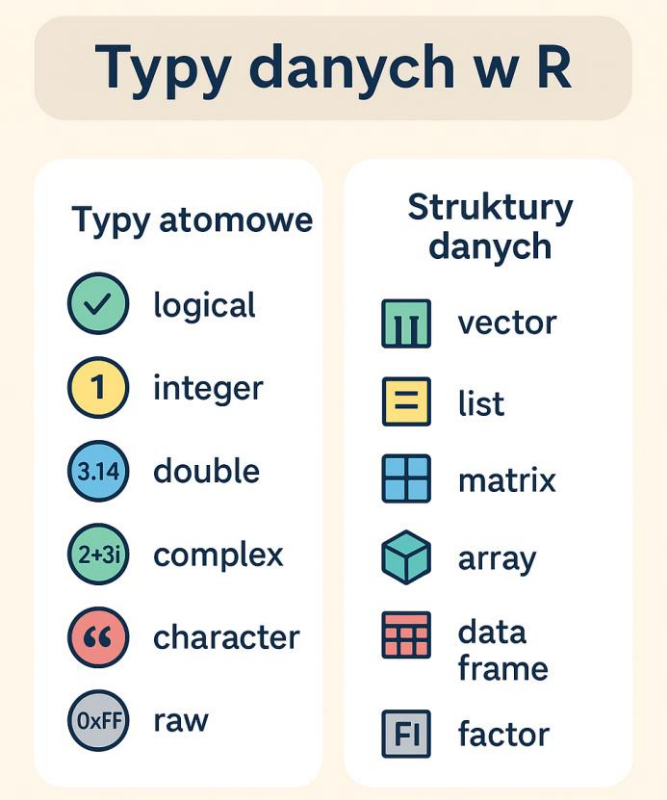


Rys. 11. Rozwiązanie przykładu 2.3.

[Tekst alternatywny. Rysunek 11 przedstawia dwa fragmenty okien programu RStudio Desktop – przykładowe rozwiązanie przykładu 2.3.: skrypt oraz konsolę R.]

3. Typy danych w R i wybrane operacje

Język R, jak każdy język programowania, operuje na danych różnych typów – liczbach, tekstach, wartościach logicznych, czy też datach. To, jakiego typu są dane, wpływa na to, co można z nimi zrobić i jak będą interpretowane przez język R. Można wyróżnić proste i złożone typy danych – rys. 12. W języku R wszystko jest obiektem – obliczenia statystyczne wykonuje się najczęściej nie na pojedynczych liczbach, ale na bardziej złożonych strukturach.



Rys. 12. Typy danych w R – rysunek poglądowy (grafikę wygenerowano z wykorzystaniem generatywnej sztucznej inteligencji w ramach licencji użytkownika)

[Tekst alternatywny. Rysunek 12 przedstawia poglądową infografikę o typach danych w R. Wyszczególnione są typy atomowe (logical, integer, double, complex, character, raw) oraz typy złożone (vector, list, matrix, array, data frame, factor).]

Przy tworzeniu niniejszego rozdziału korzystano z pomocy programu R, dokumentacji R (“Rdocumentation“, 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN (“The Comprehensive R Archive Network“, 2025) i jego podstron (gdzie znajdują się m.in. opisy poszczególnych pakietów i funkcji) oraz pozycji literaturowych (Biecek, 2018; Gągolewski, 2014;

Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012).

3.1. Wektory atomowe

Specyficzną cechą języka R jest brak rozróżnienia między wektorami, a zmiennymi składającymi się z pojedynczych wartości. Jak wspomniano już wcześniej, w R nawet jedna wartość jest wektorem – tyle że jednoelementowym.

Wektor atomowy (w skrócie: **wektor**) – uporządkowany zbiór elementów tego samego typu (istnieje więc homogeniczność). Może zawierać liczby, wartości logiczne, czy też znaki. Do utworzenia wektora (z pojedynczych elementów lub innych wektorów) służy funkcja **c()**. Do przetestowania, czy argument jest wektorem, służy funkcja **is.vector()**. Każdy wektor posiada określoną długość, tj. liczbę elementów, którą zawiera. Pierwszy element wektora posiada numer indeksu 1. Długość wektora zwraca funkcja **length()**.

Według oficjalnej dokumentacji (https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Vector-objects; choć różne źródła mogą prezentować nieco odmienne klasyfikacje), R posiada sześć podstawowych (tzw. **atomowych**) typów wektorów; są to struktury jednorodne, przechowujące dane tego samego rodzaju:

- **logical** – typ logiczny,
- **integer** – liczby całkowite,
- **double** – liczby rzeczywiste (zmiennoprzecinkowe),
- **complex** – liczby zespolone,
- **character** – napisy, łańcuchy znaków,
- **raw** – typ surowy.

Typ podstawowy wektora (z punktu widzenia R) można poznać za pomocą funkcji **typeof()**, natomiast bardziej ogólny tyb podaje funkcja **mode()** – tabela 4.

Tabela 4. Funkcje typeof(…) vs mode(…)

typeof(…)	mode(…)
logical	**logical**
integer	**numeric**
double	**numeric**
character	**character**
complex	**complex**
raw	**raw**

3.2. Wektory wartości logicznych. Podstawowe operatory relacyjne i logiczne

W R są zdefiniowane dwie stałe logiczne:

- **TRUE** (lub **T**) – prawda,
- **FALSE** (lub **F**) – fałsz.

TRUE i **FALSE** stanowią zarezerwowane słowa kluczowe języka R – w przeciwieństwie do ich skrótów: **T** oraz **F**. Dodatkowo, wielkość liter ma znaczenie – niedopuszczalne są nazwy typu **true, True**, **TruE**, **False**, **false**, **FalsE** itd.

Pojedyncza wartość logiczna jest przechowywana w postaci wektora o długości jeden. Podstawowe operatory relacyjne i logiczne zostały przedstawione w tabeli 5. Uwaga: należy pamiętać, aby nie pomylić operatora przypisania (**=**) z operatorem równości (**==**).

Tabela 5. Podstawowe operatory relacyjne i logiczne

Operator	Opis
x*==*y	***x*** równy ***y***
x*!=*y	***x*** różny od ***y***
x*<*y	***x*** mniejszy od ***y***
x*<=*y	***x*** mniejszy lub równy od ***y***
x*>*y	***x*** większy od ***y***
x*>=*y	***x*** większy lub równy od ***y***
!	Negacja
&	Logiczne AND (koniunkcja)
|	Logiczne OR (alternatywa)

Przykład 3.1.

Wektory wartości logicznych. Podstawowe operatory relacyjne i logiczne.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
5=5          #błąd
5==5         #TRUE
1!=5          #TRUE
10<10.1       #FALSE
```

```
5.51>=5.55   #FALSE
2>-1 & 2>=3 & 0!=2      #FALSE; koniunkcja (AND)
2>-1 | 2>=3 | 0!=2      #TRUE; alternatywa (OR)
typeof(5==5)            #sprawdzenie typu wektora → logical
typeof(3.44)            #sprawdzenie typu wektora → double
mode(3.44)              #sprawdzenie typu wektora → numeric
a <- T
print(a)          #TRUE
TRUE <- 200       #błąd
T <- 200          #brak błędu
> a <- T
> print(a)        #200
```

3.3. Wektory wartości liczbowych i zespolonych

W R są zdefiniowane trzy typy wektorów wartości liczbowych i zespolonych:

- całkowite (**integer**) – może on reprezentować liczby całkowite o wartościach do +/- ok. 2 miliardów (gdzie maksymalną wartość można poznać za pomocą polecenia ***.Machine$integer.max***). Wartości typu **integer** można wprowadzić tylko i wyłącznie jawnie, dopisując do liczby przyrostek **L**,
- rzeczywiste (**double**) – separatorem części dziesiętnej jest kropka,
- zespolone (**complex**) – wprowadza się je korzystając z przyrostka **i**; jednostka urojona (oznaczana jako **i**) spełnia właściwość: $\mathbf{i}^2 = -\mathbf{1}$. Każdą wartość ze zbioru liczb zespolonych można przedstawić w postaci **x** + **iy**, gdzie **x**, **y** ∈ R.

Typy **integer** oraz **double** należą do ogólnego typu liczbowego **numeric**, natomiast typ zespolony (**complex**) nie jest uznawany za typ liczbowy **numeric** (por. tabela 4).

Przy wprowadzaniu liczb można skorzystać z notacji naukowej, w której znak **e** służy do „przesuwania” kropki dziesiętnej, np. **5.98e-6**.

Podstawowe funkcje pozwalające na wykonywanie operacji na liczbach zespolonych przedstawiono w tabeli 6.

Tabela 6. Podstawowe funkcje operujące na liczbach zespolonych

Funkcja	Opis
Re(*x*)	Część rzeczywista liczby zespolonej ***x***
Im(*x*)	Część urojona liczby zespolonej ***x***
Mod(*x*)	Moduł liczby zespolonej ***x***
Arg(*x*)	Argument liczby zespolonej ***x***

Przykład 3.2.

Wektory wartości liczbowych i zespolonych.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
x <- c(2, 7, 3, 9, -1)    #utworzenie pięcioelementowego wektora x
print(x)
cat(x)
typeof(x)                 #typ double
mode(x)                   #typ numeric
y <- c(2L, 7L, 3L, 9L)    #utworzenie czteroelementowego wektora y
print(y)
typeof(y)                 #typ integer
mode(y)                   #typ numeric
z <- c(3.14e-4, 1.11, 6)  #utworzenie trzyelementowego wektora z
print(z)
typeof(z)                 #typ double
mode(z)                   #typ numeric

zesp <- -2+5i    #deklaracja liczby zespolonej zesp
print(zesp)
Re(zesp)         #część rzeczywista liczby zespolonej zesp
Im(zesp)         #część urojona liczby zespolonej zesp
Mod(zesp)        #moduł liczby zespolonej zesp
typeof(zesp)     #typ complex
mode(zesp)       #typ complex
```

3.4. Ciągi arytmetyczne. Losowanie liczby z podanego zakresu. Generowanie liczb pseudolosowych

Do generowania najprostszych ciągów arytmetycznych można wykorzystać operator **:** (dwukropek): ***wartość_startowa*** **:** ***wartość_końcowa***; krok wynosi wtedy zawsze 1 lub -1.

Ciągi arytmetyczne o dowolnej różnicy można wygenerować za pomocą funkcji:
seq(*wartość_startowa, wartość_końcowa, by=* *lub* ***length.out*=)**
gdzie: ***wartość_startowa*** oznacza pierwszy wyraz ciągu, ***wartość_końcowa*** oznacza ostatni wyraz ciągu, natomiast ***by*** oznacza wielkość różnicy (krok; parametr

domyślny), a zamiennie ***length.out*** odpowiada za utworzenie wektora o zadanej długości.

Na wylosowanie pewnej liczby wartości ***n*** z podanego zakresu (będącego wektorem), bez (***replace = FALSE***) lub z powtórzeniami (***replace = TRUE***) pozwala funkcja: **sample(*wektor, size = n, replace = TRUE/FALSE*)**.

Do generowania liczb pseudolosowych (ponieważ „emulują" one losowość w określony sposób) służy funkcja **runif(*n*)**. Do właściwego działania wymaga ona podania jednego argumentu ***n***, tj. długości wektora wynikowego. Domyślnie losowane są liczby z przedziału <0, 1>. Opcjonalnie można podać również argumenty ***min*** i ***max***, wtedy losowane będą liczby z zakresu **<*min*, *max*>**: **runif(*n, min, max*)**.

Funkcje generujące liczby pseudolosowe wykorzystują tzw. **ziarno** (ang. *seed*), które determinuje ciąg kolejnych otrzymywanych wartości. Przed funkcją taką można wywołać polecenie **set.seed()**, które jest wykorzystywane, by ustawić „ziarno" generatora liczb losowych R na konkretną wartość, co umożliwia tworzenie tego samego (a więc identycznego) zestawu „losowych" liczb za każdym razem, gdy skrypt jest wywoływany – bez względu na wykorzystywany komputer, system operacyjny, czy też porę dnia.

W ogólności, w wielu różnych zastosowaniach (np. modelowanie złożonych zjawisk) istnieje konieczność generowania liczb losowych; R posiada wiele możliwości generowania takich wartości z różnych rozkładów. W razie potrzeby można więc zmienić generator liczb pseudolosowych na inny, wykorzystując funkcję **RNGkind()**. W dostępnych materiałach w sieci Internet można znaleźć wiele informacji na ten temat.

Przykład 3.3.

Ciągi arytmetyczne. Losowanie liczby z podanego zakresu. Generowanie liczb pseudolosowych.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
c(-5:5)          #ciąg arytmetyczny od -5 do 5 (krok domyślny = 1)
c(5:-5)          #ciąg arytmetyczny od 5 do -5 (krok domyślny = -1)
c(0.5:3.5)       #ciąg arytmetyczny liczb zmiennoprzecinkowych
```

```
seq(2, 30, by=4)          #ciąg od 2 do 30, krok = 4
seq(10, 0, by=-2)         #ciąg od 10 do 0, krok ujemny (-2)
seq(1, 5, length.out=3)   #trzyelementowy ciąg arytmetyczny
w <- seq(1, 12, 3)        #sekwencja przypisana do zmiennej
print(w)
typeof(w)

sample(1:10, size=4, replace=TRUE)   #losowanie 4 liczb z powtórzeniami
sample(1:10, size=4, replace=FALSE)  #losowanie 4 liczb bez powtórzeń
     #uruchom poniższe 2 linie kodu wielokrotnie, co zauważyłeś?
set.seed(082025)
sample(1:500, size=2, replace=TRUE)
     #zawsze "losowane" są te same wartości!

runif(1)                #wylosowanie jednej liczby z przedziału (0,1)
runif(5)                #wylosowanie pięciu liczb z przedziału (0,1)
runif(5, min = 10, max = 20)     #wylosowanie 5 liczb z przedziału (10,20)
x <- runif(1000)        #wylosowanie 1000 liczb z przedziału (0,1)
mean(x)                 #średnia arytmetyczna wylosowanych wartości
min(x); max(x)          #najmniejsza i największa wylosowana wartość
     #uruchom poniższe 2 linie kodu wielokrotnie, co zauważyłeś?
set.seed(082025)
runif(1)
```

3.5. Wektory napisów

Tabela 7. Wybrane znaki specjalne

Znak specjalny	Opis
\n	Nowy wiersz
\r	Powrót karetki do początku wiersza
\t	Tabulator
\b	Usunięcie znaku poprzedzającego (*backspace*)
****	Ukośnik (*backslash*) \
\'	Apostrof '
\"	Cudzysłów "

Wartościami obiektów typu **character** są napisy (łańcuchy znaków), w których mogą występować dowolne znaki (w tym także tzw. znaki specjalne – tabela 7). W języku R wprowadzanie napisów należy rozpocząć znakiem **'** lub **"**, przy czym początek i koniec napisu muszą być koniecznie oznaczone tym samym znakiem. Na łańcuchach znaków można wykonywać różne operacje, takie jak wyodrębnianie

fragmentów, łączenie kilku napisów w jeden, wyszukiwanie określonych podciągów i wiele innych form przekształceń tekstu. R, pomimo że napis jest ciągiem znaków, typ **character** reprezentuje jako wektor całych napisów, a więc „ciąg ciągów znaków”. Do zliczania liczby znaków w łańcuchu służy funkcja **nchar()**.

Do wyświetlania napisów (w ogólności: obiektów) służą funkcje:

- **cat()** – konwertuje/usuwa niektóre informacje specyficzne dla kodu (np. interpretuje znaki specjalne, przykładowo **\n** jest konwertowane na znak nowej linii); zwraca ponadto wartość **NULL**,
- **print()** – jest domyślną funkcją używaną przez R do prezentacji wyników w konsoli; zwraca wynik operacji jako obiekt danych.

Obie powyższe funkcje wyświetlają wartość obiektu, jednak w odmienny sposób.

Do sklejania (łączenia) wektorów napisów służy funkcja **paste()**. Argument ***sep= " "*** służy do określania separatorów, którymi sklejane są wektory wejściowe. Jeśli określi się argument ***collapse= " "*** , wtedy elementy tego wektora zostaną połączone. Istnieje także funkcja **paste0()**, w której domyślnym separatorem jest ***""***, a więc pozwala ona na szybkie łączenie napisów bez spacji.

Aby wyświetlić określone dane stosując przy tym pożądane formatowanie, przydatna jest funkcja **format()**. Służy ona do konwersji określonego obiektu na typ znakowy, przy czym ponadto pozwala zdefiniować formatowanie (np. ile miejsc dziesiętnych ma zostać wypisanych w przypadku liczby, czy tekst ma zostać wyjustowany do lewej, czy do prawej albo, czy ma być wykorzystana notacja naukowa itd.). Składnia:
format(*x*, *digits*, *nsmall*, *scientific*, *width*, *justify = c(“left”, “right”, “center”, “none”)*)
gdzie: ***x*** – wektor wejściowy; ***digits*** – całkowita liczba wyświetlanych cyfr; ***nsmall*** – minimalna liczba cyfr po prawej stronie separatora dziesiętnego; ***width*** – minimalna szerokość pola, która ma być wyświetlana, ***justify*** – wyjustowanie tekstu.

Przykład 3.4.

Wektory napisów.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
s <- "R \n to \n \'język\'"     #\n to znak specjalny – nowa linia
print(s)                        #co ze znakami specjalnymi?
cat(s)                          #cat() interpretuje znaki specjalne
```

```
s1 <- "R"
s2 <- "jest"
s3 <- "super"
paste(s1, s2, s3)             #łączenie wektorów s1, s2, s3
paste0(s1, s2, s3)            #łączenie wektorów bez separatora
paste(s1, s2, s3, sep=";")    #łączenie wektorów z separatorem ";"
typeof(s)                     #typ character
mode(s)                       #typ character
s4 <- paste(s1, s2, s3)
print(s4)
s5 <- c(s1, s2, s3)           #utworzenie trzyelementowego wektora
print(s5)
length(s4)                    #wektor s4 zawiera 1 element
length(s5)                    #wektor s5 zawiera 3 elementy
nchar(s4)                     #liczba znaków w napisie s4
nchar(s5)                     #liczba znaków w każdym elemencie wektora s5

Tekst <- "ID"
X <- 78.56125
format(Tekst, width = 20)
format(Tekst, width = 10, justify = "right")
format(Tekst, width = 10, justify = "centre")
format(X, scientific = TRUE)      #notacja naukowa
format(X, digits = 5)             #całkowita liczba wyświetlanych cyfr: 5
format(X, digits = 7)             #całkowita liczba wyświetlanych cyfr: 7
```

3.6. Pozostałe typy danych

W niniejszym rozdziale krótko i zbiorczo zostaną przedstawione pozostałe, lecz istotne z punktu widzenia *inżynierii danych*, typy danych w języku R.

W analizie danych często zachodzi potrzeba, aby oznaczyć pewne wartości jako nieznane lub niedostępne; w statystyce mówi się o brakach danych (ang. *missing data*). Do reprezentacji braków danych w języku R służy obiekt **NA** (ang. *not available*). Do sprawdzenia, które elementy wektora są typu **NA** służy funkcja **is.na()**; zwraca ona **TRUE**, jeśli element jest typu **NA** lub **FALSE** – w przeciwnym przypadku. Uwaga: większość operacji na wartościach **NA** daje w wyniku również wartość **NA**.

W wyniku wykonywania pewnych działań na wartościach liczbowych może się zdarzyć, że w ich wyniku uzyskana zostanie „nie-liczba”(obiekt **NaN** – ang. *not*

a number, wartość nieokreślona) lub wartość nieskończona (**Inf** lub **–Inf** – ang. *infinity*), lecz ich znaczenie jest inne niż braków danych.

W R istnieje także obiekt **NULL**, który może być np. używany przez funkcje, gdy nie zwracają one żadnej konkretnej wartości. Do sprawdzenia, czy dana wartość jest typu **NULL** służy funkcja **is.null()**. Uwaga: nie należy mylić wartości **NA** oraz **NULL** – pierwsza z nich oznacza brak w danych i może stanowić dowolny element każdego wektora.

Zmienne jakościowe w R, które przyjmują określoną liczbę wartości, najczęściej nie liczbowych, są reprezentowane przez typ czynnikowy (wyliczeniowy) **factor** – np. **kolor oczu**(***zielone***, ***niebieskie***, ***brązowe***). Zmienne te można dodatkowo oznaczyć jako uporządkowane – dodaje im się wtedy klasę **ordered**.

Przykład 3.5.

Pozostałe typy danych w R.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
2/0                         #dodatnia nieskończoność Inf
-5/0                        #ujemna nieskończoność -Inf
0/0                         #NaN – nie liczba
is.nan(0/0)                 #TRUE
sqrt(-4)                    #NaN – pierwiastek z liczby ujemnej
log(0)                      #-Inf – logarytm z zera
is.finite(2/0)              #czy wartość jest skończona?
x <- c(2, 5, NA, 8, NA)     #wektor x zawiera dwie wartości NA
print(x)
is.na(x)                    #wartości TRUE dla braków danych
mean(x)                     #wynik: NA (średnia nie jest obliczona)
mean(x, na.rm=TRUE)         #pominięcie braków w danych

w <- NULL
print(w)
is.null(w)                  #TRUE
length(w)                   #NULL; nie ma elementów
length(NA)                  #NA to brak w danych, ale długość = 1
```

```
      #typ factor
kolory <- factor(c("zielone", "niebieskie", "brązowe", "niebieskie"))
print(kolory)
levels(kolory)            #poziomy (dostępne kategorie)
nlevels(kolory)           #liczba poziomów, tu: 3

      #typ factor uporządkowany
oceny <- ordered(c("dobry", "dostateczny", "bardzo dobry", "dobry"),
                 levels=c("dostateczny", "dobry", "bardzo dobry"))
print(oceny)
```

3.7. Podstawowe operacje na wektorach. Wektoryzacja operacji

Jak już wspomniano powyżej, do utworzenia wektora (z pojedynczych elementów lub innych wektorów) służy funkcja **c()** (od ang. *combine* – złącz).

Indeksowanie wektorów polega na możliwości wyboru określonych elementów wektora przez wskazanie numeru(ów) ich indeksów (pozycji). Indeksy wektorów w R numerowane są od **1** do ***n*** (gdzie ***n*** oznacza długość, tj. liczbę elementów wektora). Do poszczególnych elementów wektora można odwoływać się za pomocą indeksów: **nazwa_wektora[*zakres*]**. Długość wektora zwraca funkcja **length()**.

Wektory można także tworzyć przez replikację (a więc powtórzenie) za pomocą funkcji **rep(*wektor, liczba_powtórzeń*)**.

Złączenie dwóch wektorów jest wektorem. Wektory atomowe przechowują elementy jednego, ściśle określonego typu, mają więc jednorodną strukturę, choć wyjątek stanowi możliwość umieszczenia w wektorze jednej lub wielu wartości **NA**. Gdy łączone są ze sobą wektory różnych typów, R uzgadnia finalny typ wektora tak, by informację o najbardziej ogólnym z obiektów można było przechowywać bez znaczącej straty (choć konwersja liczb rzeczywistych na napisy może czasem powodować ich zaokrąglenie) – ta właściwość nazywa się uzgadnianiem typów (**koercją**).

Wektory w R posiadają szczególny status – są czymś więcej niż tylko prostymi kontenerami, ponieważ język ten jest **zwektoryzowany**. Oznacza to, że wiele operacji wykonywanych jest automatycznie na wszystkich elementach wektora (typ **vector**), bez konieczności ich jawnego iterowania w pętli (jak ma to miejsce w wielu innych językach programowania). Wektory w R nie posiadają wymiarów, przez co nie istnieje coś takiego jak wektor kolumnowy lub wektor wierszowy (choć można je

przedstawić jako macierze jednowymiarowe). W związku z tym typ **vector** nie jest odpowiednikiem matematycznego pojęcia wektora. Działania na dwóch liczbach dają w wyniku liczbę. Działania na dwóch wektorach tej samej długości – wektor tej samej długości, przy czym operacje wykonywane są „element po elemencie". Jeżeli argumentami działania są wektory o różnej długości, to krótszy z nich jest replikowany (powielany) tyle razy, ile potrzeba, aby był większy lub równy dłuższemu; jeśli po replikacji długości obu argumentów się nie zgadzają, to działanie jest wykonywane, ale R generuje ostrzeżenie. W przypadku obiektów wielowymiarowych, aby operacja była wykonalna, wymagane jest dopasowanie wszystkich ich wymiarów.

Wybrane funkcje pozwalające na wyświetlanie określonych elementów wektora, a także na manipulowanie jego elementami, przedstawiono w tabeli 8. Tabela 9 zawiera wybrane funkcje pozwalające na sprawdzenie typu wektora, zaś tabela 10 funkcje pozwalające na konwersję typu wektora, a tabela 11 wybrane funkcje agregujące (zwracające pojedynczą wartość, w pewnym sensie charakterystyczną dla całego danego wektora).

Tabela 8. Wybrane funkcje operujące na wektorach

Funkcja	Opis
***w*[e]**	Trzeci element wektora ***w***
***w*[2:4]**	Od drugiego do czwartego elementu wektora ***w***
***w*[c(1,4,7)]**	Elementy wektora ***w*** na pozycjach o numerach indeksu 1, 4 i 7
***w*[-c(2,3)]**	Wszystkie elementy wektora ***w*** oprócz 2 i 4
***w*[w>5]**	Elementy wektora ***w*** większe niż 5
***w*[length(*w*)]**	Ostatni element wektora ***w***
***w*[*w* %% 2 == 0 & *w* > 6]**	Wypisuje elementy wektora ***w***: parzyste i większe niż 6
***w*[1]<-44**	Wstawienie wartości 44 na pierwszą pozycję wektora ***w*** (aktualna wartość zostanie nadpisana!)
length(*w*)	Liczba elementów wektora ***w***
sort(*w*)	Sortowanie elementów wektora ***w*** (rosnąco)
rev(*w*)	Sortowanie elementów wektora ***w*** (malejąco)
which(*w*>=3)	Podaje numery indeksów elementów wektora ***w***, które są większe lub równe 3
which.min(*w*)	Podaje numer indeksu najmniejszego elementu wektora ***w***
which.max(*w*)	Podaje numer indeksu największego elementu wektora ***w***

Tabela 9. Wybrane funkcje pozwalające na sprawdzenie typu wektora

Funkcja	Opis
is.numeric()	Test, czy argument jest liczbą
is.integer()	Test, czy argument jest liczbą całkowitą
is.double()	Test, czy argument jest liczbą rzeczywistą
is.complex()	Test, czy argument jest liczbą zespoloną
is.logical()	Test, czy argument jest wartością logiczną
is.character()	Test, czy argument jest znakiem lub napisem
is.factor()	Test, czy argument jest typu wyliczeniowego
is.atomic()	Test, czy argument jest typu atomowego
is.na()	Test, czy elementy wektora są typu **NA** lub, w przypadku wektorów typów numeric i complex, **NaN**
is.finite()	Test, czy elementy wektora liczbowego nie są równe **NA**, **NaN**, **Inf** ani **–Inf**

Tabela 10. Wybrane funkcje pozwalające na konwersję typu wektora

Funkcja	Opis
as.numeric()	Konwersja na typ liczbowy
as.integer()	Konwersja na wartość całkowitoliczbową
as.double()	Konwersja na wartość rzeczywistą
as.complex()	Konwersja do liczby zespolonej
as.logical()	Konwersja na wartość logiczną
as.character()	Konwersja na typ znakowy
as.factor()	Konwersja na typ wyliczeniowy
as.list()	Konwersja do listy
unlist()	Konwersja z listy do wektora
as.matrix()	Konwersja na typ macierzowy
as.data.frame()	Konwersja na ramkę danych

Tabela 11. Wybrane funkcje agregujące

Funkcja	Opis
sum(*w*)	Suma elementów wektora ***w***
prod(*w*)	Iloczyn elementów wektora ***w***
mean(*w*)	Średnia arytmetyczna elementów wektora **w**
var(*w*)	Wariancja próbkowa
sd(*w*)	Odchylenie standardowe
min(*w*)	Wartość minimalna
max(*w*)	Wartość maksymalna
median(*w*)	Mediana („wartość środkowa”)
quantile(*w*, *p*)	Kwantyl rzędu ***p***

Aby „pozbyć się" brakujących wartości, można użyć m.in. funkcji:

- **na.omit()** – zwraca wektor pozbawiony elementów **NA**; dane wyjściowe zawierają jednak także pewne dodatkowe informacje – aby je usunąć, w przypadku wartości numerycznych, należy wykorzystać funkcję **as.numeric()**,
- **complete.cases()** – tworzy wektor wartości logicznych tej samej długości co argument wejściowy z wartościami **TRUE** dla elementów bez braków danych i **FALSE** dla tych zawierających **NA**.

W celu wyeliminowania duplikatów, można wykorzystać poniższe funkcje:

- **unique()** – zwraca obiekt tego samego typu (np. wektor) co argument wejściowy, ale z usuniętymi powtarzającymi się elementami; wyodrębnia więc tylko niepowtarzające się elementy,
- **duplicated()** – tworzy wektor wartości logicznych tej samej długości co argument wejściowy, którego elementy przyjmują wartość **TRUE** w przypadku wykrycia powtórzeń, a **FALSE** – gdy pojawiają się po raz pierwszy.

Wektory zawierające więcej niż jeden element również można porównywać między sobą w języku R. Wynikiem takiej operacji jest wektor o identycznej długości, którego każdy element zawiera wartość logiczną **TRUE** bądź **FALSE**. Aby sprawdzić, czy wszystkie wynikowe elementy mają wartość **TRUE** można wykorzystać funkcję **all()**, natomiast, aby sprawdzić, czy którykolwiek z elementów przyjmuje wartość **TRUE** – funkcję **any()**.

Przykład 3.6.

Podstawowe operacje na wektorach. Wektoryzacja operacji.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
v <- 1:12          #utworzenie wektora liczbowego z wartościami od 1 do 12
print(v)
print(v[0])            #numeracja elementów rozpoczyna się od 1!
print(v[1])            #pierwszy element wektora
print(v[4])            #czwarty element wektora
print(v[c(2, 6, 9)])   #kilka wybranych elementów wektora
print(v[2:5])          #elementy wektora od drugiego do piątego
print(v[-c(3, 7)])     #wszystkie elementy wektora oprócz 3 i 7
print(v[length(v)])    #ostatni element wektora
```

```
which(v > 7)            #numery indeksów elementów większych od 7
which.min(v)            #numer indeksu najmniejszego elementu wektora
which.max(v)            #numer indeksu największego elementu wektora
v2 <- v * 3             #każdy element wektora pomnożony przez 3
print(v2)
v3 <- v + 10            #do każdego elementu wektora dodano 10
print(v3)
v4 <- v^2               #podniesienie każdego elementu wektora do kwadratu
c(0, 1) * v             #tzw. reguła zawijania; automatyczna replikacja
                        #wektora c(0, 1)
rep_v <- rep(v, 2)      #dwukrotna replikacja wektora v
print(rep_v)
length(rep_v)           #długość powstałego wektora – 24 elementy
sort(v)                 #sortowanie elementów wektora rosnąco
rev(v)                  #sortowanie elementów wektora malejąco
      #sprawdzanie własności wektora v
is.numeric(v)
is.character(v)
is.atomic(v

v5 <- c(1, 2, "trzy", 4)      #uzgadnianie typów (koercja)
print(v5)                     #liczby zamienione na napisy
typeof(v5)                    #typ character

v6 <- c(1, 2, 2, 3, 3, 3, 4)
unique(v6)                    #usunięcie duplikatów
duplicated(v6)               #wartości TRUE dla powtarzających się elementów
6[v6 == 2] <- 100       #zamiana wszystkich wartości 2 w wektorze v6 na 100
print(v6)
v7 <- c(5, 10, 15)
v8 <- c(5, 20, 15)
v7 == v8                 #porównywanie element po elemencie
all(v7 == v8)            #czy wszystkie elementy wektorów są równe?
any(v7 == v8)            #czy chociaż jeden element wektorów jest równy?
```

3.8. Macierze

Macierz posiada istotne znaczenie w matematyce i statystyce. W języku R jest to typ złożony (**matrix**), który stanowi dwuwymiarowa struktura, reprezentowana przez wektory atomowe z ustawionym atrybutem specjalnym ***dim***. Ma postać prostokątnej tablicy składającej się z wierszy i kolumn, w której wszystkie elementy muszą być

tego samego typu (najczęściej numeryczne). Indeksowanie wierszy i kolumn rozpoczyna się od 1. Do tworzenia macierzy służy funkcja:
matrix*(wartości, nrow = liczba_wierszy, ncol = liczba_kolumn)*.

Pierwszym argumentem funkcji **matrix()** jest wektor, który stanowi zestaw wartości wypełniających jej zawartość. Kolejne argumenty tej funkcji określają wymiary macierzy, tj. liczbę wierszy (***nrow***) oraz liczbę kolumn (***ncol***). Domyślnie dane umieszczane są kolumnami (od góry do dołu), począwszy od pierwszej kolumny po lewej stronie. Można jednak zdefiniować parametr ***byrow = TRUE***, a wtedy wartości będą zapisywane do kolejnych wierszy zamiast kolumn.

Dostęp do wybranego elementu macierzy jest możliwy poprzez podanie numeru wiersza i kolumny w nawiasie kwadratowym, np. **macierz[*i, j*]**. W przypadku wykonywania podstawowych działań na poziomie poszczególnych elementów (dodawanie, odejmowanie, mnożenie, dzielenie itd.), macierze działają podobnie jak wektory.

Macierze domyślnie nie mają przypisanych nazw wierszy ani kolumn. Można je jednak dodać, korzystając z funkcji **rownames()** (dla wierszy) oraz **colnames()** (dla kolumn).

Wybrane funkcje pozwalające na wykonywanie operacji na macierzach przedstawiono w tabeli 12.

Tabela 12. Wybrane funkcje operujące na macierzach

Funkcja	Opis
sum(*m*)	Suma elementów macierzy ***m***
rowSums(*m*)	Suma elementów w poszczególnych wierszach macierzy ***m***
colSums(*m*)	Suma elementów w poszczególnych kolumnach macierzy ***m***
dim(*m*)	Wymiar macierzy ***m***
ncol(*m*)	Liczba kolumn macierzy ***m***
nrow(*m*)	Liczba wierszy macierzy ***m***
t(*m*)	Transpozycja macierzy ***m***
m1*%*%*m2	Iloczyn macierzy ***m1*** i ***m2*** (uwaga: liczba kolumn macierzy ***m1*** musi być równa liczbie wierszy macierzy ***m2***)
solve(*m*)	Macierz odwrotna do macierzy ***m*** (uwaga: macierz musi być kwadratowa)
det(*m*)	Wyznacznik macierzy kwadratowej ***m***
prod(*m*)	Iloczyn elementów macierzy ***m***
min(*m*)	Wartość minimalna
max(*m*)	Wartość maksymalna

Funkcja	Opis
cbind(*m*, *k*)	Tworzy macierz poprzez rozbudowanie macierzy ***m*** o kolumnę ***k*** (dołączanie kolumn)
rbind(*m*, *w*)	Tworzy macierz poprzez rozbudowanie macierzy ***m*** o wiersz ***w*** (dołączanie wierszy)
diag(*m*)	Tworzenie macierzy diagonalnej
colMeans(*m*)	Średnia arytmetyczna elementów w poszczególnych kolumnach macierzy ***m***
rowMeans(*m*)	Średnia arytmetyczna elementów w poszczególnych wierszach macierzy ***m***
***m*[,3]**	Trzecia kolumna macierzy ***m***
***m*[3,]**	Trzeci wiersz macierzy ***m***
***m*[1,4]**	Elementy macierzy ***m*** z 1 wiersza i 4 kolumny
***m*[4:5,]**	Czwarty i piąty wiersz macierzy ***m***
***m*[c(1,3),4:6]**	Pierwszy i trzeci wiersz oraz od czwartej do szóstej kolumny macierzy ***m***

Przykład 3.7.

Macierze.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
M1 <- matrix(1:12, nrow = 4, ncol = 3)     #tworzy macierz 4x3 wypełnioną
                            #wartościami od 1 do 12 (domyślnie kolumnami)
M2 <- matrix(seq(10, 21), nrow = 3, ncol = 4, byrow = TRUE)    #tworzy
            #macierz 4x3 wypełnioną wartościami od 10 do 12 (wierszami)
rownames(M2) <- c("w1", "w2", "w3")       #nazwy wierszy
colnames(M2) <- paste0("k", 1:4)          #nazwy kolumn
is.matrix(M2)      #czy jest macierzą? TRUE
is.array(M2)       #czy jest tablicą? TRUE
is.vector(M2)      #czy jest wektorem? FALSE
nrow(M2)           #liczba wierszy macierzy
ncol(M2)           #liczba kolumn macierzy
print(M2[,5])      #piąta kolumna macierzy M2 – błąd!
print(M2[1,])      #pierwszy wiersz macierzy
print(M2[2,3])     #element macierzy M2 z 2 wiersza i 3 kolumny
T_M2 <- t(M2)      #macierz transponowana
print(T_M2)
O_M2 <- solve(M2)  #macierz odwrotna – błąd!
sum(M2)            #suma elementów macierzy
rowSums(M2)        #suma elementów w poszczególnych wierszach macierzy
prod(M2)           #iloczyn elementów macierzy
min(M2)            #minimalny element macierzy
```


Politechnika Świętokrzyska Kielce University of Technology
Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki” nr FERS.01.05-IP.08-0234/23

```
max(M2)            #maksymalny element macierz
det(matrix(c(2, 3, 1, 4), nrow = 2))      #wyznacznik macierzy 2x2
```

3.9. Tablice

W języku **R** tablica (typ złożony **array**) jest uogólnieniem macierzy – każdą macierz można traktować jako dwuwymiarową tablicę. Tablice są więc strukturami **wielowymiarowymi**, które mogą mieć dowolną (skończoną) liczbę wymiarów. Podobnie jak w przypadku wektorów i macierzy, wszystkie elementy tablicy muszą należeć do tego samego typu. Jednakże konstrukcje o większej liczbie wymiarów niż 2 są używane stosunkowo rzadko. Do tworzenia tablic służy funkcja **array()**, a numeracja indeksów zaczyna się od **1**. Aby odwołać się do konkretnego elementu tablicy, stosuje się nawiasy kwadratowe, przekazując wektor indeksów odpowiadających kolejnym wymiarom.

Przykład 3.8.

Tablice.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
     #tablica 3-wymiarowa T1: 2 x 3 x 4; wypełniona liczbami od 1 do 24
T1 <- array(1:24, dim = c(2, 3, 4))
print(T1)
print(T1[ , , 2])        #wyświetla całą 2. warstwę tablicy (macierz 2x3)
print(T1[1, 2, 3])
dimnames(T1) <- list(               #nadawanie nazw wymiarom
  wiersze = c("r1", "r2"),
  kolumny = c("c1", "c2", "c3"),
  warstwy = paste0("w", 1:4)
)
print(T1)
is.matrix(T1)      #czy jest macierzą? FALSE
is.array(T1)       #czy jest tablicą? TRUE
is.vector(T1)      #czy jest wektorem? FALSE
```

3.10. Listy

W trakcie wykonywania obliczeń często otrzymuje się wynik, który nie ogranicza się do pojedynczej wartości, tylko stanowi obiekt zawierający wiele informacji. Do przechowywania takich danych w języku R służą **listy** (typ złożony **list**), które mogą zawierać dowolną liczbę elementów i to elementów o różnych typach: wartości liczbowe, teksty, ramki danych, inne listy itd. Listę tworzy się przy pomocy funkcji **list()**, a każdy argument przekazany do tej funkcji staje się jednym z elementów listy (może nim być nawet kolejna lista). Numeracja elementów rozpoczyna się od **1**.

Do wybranego elementu listy można się odwołać poprzez podwójne nawiasy kwadratowe: **lista[[*nr_indeksu*]]**. Liczbę elementów listy sprawdzić można funkcją **length()**. Ponadto, poszczególne elementy mogą mieć własne **nazwy**, co pozwala odwoływać się do nich przy pomocy operatora $, np. **lista$nazwa**; aktualne nazwy elementów wyświetla polecenie **names()**. Po uzyskaniu dostępu do elementu listy można go traktować zgodnie z typem danych tego elementu – dozwolone jest np. zagnieżdżone indeksowanie elementów. Nowe elementy listy można dodawać z nazwą lub bez nazwy:

lista[[nr_indeksu]] <- wartość
lista[[nazwa_elementu]] <- wartość

Przykład 3.9.

Listy.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
L1 <- list(                          #utworzenie listy trzyelementowej
  liczby = c(5, 10, 15),
  macierz = matrix(1:6, nrow = 2),
  tekst = "Analiza danych"
)
print(L1)
print(L1[[2]])       #drugi element listy
print(L1[[2]][1,])   #drugi element listy (macierz) i jego pierwszy wiersz
L1$nowy <- c(TRUE, FALSE, TRUE)    #dodanie nowego element do listy
print(L1)
str(L1)        #szczegółowe informacje o liście
el <- L1[[2]][3]
```

```
print(el)
typeof(el)
is.list(L1)       #TRUE
is.list(el)       #FALSE
```

3.11. Daty

W R do przechowywania samej daty (dzień, miesiąc, rok) stosuje się typ **Date**, natomiast do przechowywania zarówno daty, jak i czasu – typ **POSIXct**. Oba typy reprezentowane są wewnętrznie jako liczby: **Date** jako liczba dni od 1 stycznia 1970 roku, a **POSIXct** jako liczba sekund od tej samej daty.

Aktualną datę można pozyskać funkcją **Sys.Date()**, a bieżącą datę i czas funkcją **date()**. Aby przekształcić tekstową reprezentację daty (typ **character**) na obiekt typu **Date**, można wykorzystać funkcję **as.Date()**. Domyślny format to yyyy-mm-dd. Analogicznie, w celu konwersji daty i czasu zapisanego w postaci tekstu (typ **character**) na obiekt typu **POSIXct**, wykorzystać można funkcję **as.POSIXct()**. Domyślny format to **yyyy-mm-dd hh:mm:ss**.

Przykład 3.10.

Daty.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
dzis <- Sys.Date()     #pobiera dzisiejszą datę oraz bieżący czas
czas <- date()         #pobiera bieżący czas
print(dzis)
print(czas)
typeof(dzis)           #double
typeof(czas)           #character
as.numeric(dzis)
data1 <- as.Date("2025-09-01")    #obiekt typu Date z napisu
print(data1)
typeof(data1)
class(data1)       #Date

     #tworzenie obiektu typu POSIXct z datą i czasem
data2 <- as.POSIXct("2025-09-01 14:30:00")   #obiekt typu POSIXct z napisu
print(data2)
```

```
typeof(data2)
class(data2)       #PoSIXt
```

3.12. Ramki danych

Ramki danych (**data.frame**) należą do najważniejszych struktur danych w języku R. W nawiązaniu do studiów na kierunku *inżynieria danych*, stanowią one niejednokrotnie fundament przy pracy z danymi w trakcie ich analizy. Są to obiekty, w których dane przechowywane są w postaci macierzowej (tj. w układzie kolumn i wierszy) – przypominają więc zbiory danych znane z arkuszy kalkulacyjnych, Kolumny odpowiadają **zmiennym** (ang. *variables*), a wiersze **obserwacjom** (ang. *observations*). Każda kolumna jest wektorem (typ **vector**) i wszystkie kolumny mają jednakową długość. Poszczególne kolumny mogą przechowywać dane różnych typów, natomiast w obrębie jednej kolumny wszystkie wartości muszą być dokładnie tego samego typu.

Ramki danych w R przechowywane są jako listy, których elementy są wektorami atomowymi o jednakowej długości. Tworzy się je za pomocą funkcji **data.frame()**. W narzędziu RStudio Desktop możliwe jest wygodne przeglądanie zawartości ramki danych w zakładce **Workspace**.

Wiersze ramki danych mogą być nazywane przy użyciu funkcji **row.names()**, której argumentem jest wektor unikalnych wartości identyfikujących każdy wiersz. Analogicznie, nazwy kolumn można sprawdzać i zmieniać w razie potrzeb za pomocą funkcji **colnames()** lub **names()**. Ponieważ ramka danych jest listą, dopuszczalne jest stosowanie operacji indeksowania charakterystycznych dla list, w tym podwójnych nawiasów kwadratowych **([[ ]]**), podobnie jak w przypadku macierzy. Wybranie kolumny możliwe jest także za pomocą operatora **$**, poprzedzonego nazwą ramki danych; po znaku **$** należy podać nazwę kolumny. Aby zbadać, ile kolumn zawiera ramka danych, wykorzystuje się funkcję **length()**.

Podobnie jak w wektorach, do filtrowania danych można używać operatorów indeksowania. Istnieje także funkcja **subset()**, która pozwala zwrócić podzbiór ramki danych, wraz z argumentem ***select*** umożliwiającym wybór kolumn (nie wymaga on używania cudzysłowów bądź apostrofów).

Transformacje (a więc przekształcenia) danych w kolumnach można przeprowadzać za pomocą funkcji:

- **with()** – wykonuje żądane wyrażenia na danych bez modyfikowania oryginalnej ramki danych,

- **within()** – wykonuje żądane wyrażenia i tworzy kopię ramki danych; w razie potrzeby zapamiętania zmian, można nadpisać oryginalną ramkę danych jej zmodyfikowaną wersją.

Alternatywnie, do transformacji można wykorzystać funkcję **transform()**, pozwalającą modyfikować dane w istniejących kolumnach lub dodawać nowe. Ogólnie, R udostępnia wiele sposób przeprowadzania transformacji danych. Nowe kolumny i wiersze można również dodawać przy użyciu funkcji **cbind()** i **rbind()**, z uwzględnieniem możliwej konwersji typów.

Do sortowania danych względem kolumn służy funkcja **sort()**, natomiast agregację danych w pewnych podgrupach realizuje funkcja **aggregate()** – dzieli ona obserwacje na podgrupy, a obliczenia wykonywanie są niezależnie w każdej z tych podgrup: **aggregate(*x*, *by*, *FUN* [, …])**
gdzie: ***x*** – obiekt R (tu ramka danych); ***by*** – lista elementów grupujących, z których każdy jest tak długi, jak zmienne w ramce danych ***x***; ***FUN*** – funkcja do obliczania statystyk podsumowujących, które można zastosować do wszystkich podzbiorów danych.

Istnieje także funkcja **ave()**, która pozwala obliczać wartości w podgrupach, ale podaje je oddzielnie dla każdego przypadku.

W języku R istnieją także funkcje służące do szybkiego podejrzenia fragmentu danych (np. ramki danych):

- **head(*x*, *n*)** – zwraca pierwsze ***n*** wierszy obiektu ***x*** (domyślnie 6),
- **tail(*x*, *n*)** – zwraca ostatnie ***n*** wierszy obiektu ***x*** (domyślnie 6).

3.13. Obiekty – sprawdzanie i modyfikowanie właściwości.

W trakcie pracy z R, pomocne mogą być także pewne funkcje pozwalające na sprawdzanie i modyfikowanie właściwości obiektów. Przedstawiono je w tabeli 13.

Tabela 13. Wybrane funkcje pozwalające na sprawdzanie i modyfikowanie właściwości obiektów

Funkcja	Opis
fix(*obiekt*)	Uruchamia edytor pozwalający na modyfikację wartości wybranego obiektu – np. zmiennej, funkcji, czy też ramki danych. Funkcja ta umożliwia wprowadzanie zmian bez konieczności przepisywania całego kodu funkcji lub ręcznej edycji danych w ramce

Funkcja	Opis
class(*obiekt*)	Zwraca nazwę klasy obiektu. Nazwę klasy można zmieniać bez zmiany zawartości obiektu. Różne funkcje mogą zachowywać się odmiennie dla obiektów o tej samej zawartości, lecz różnej klasie
unclass(*obiekt*)	Zwraca obiekt pozbawiony atrybutu **class**. Usuwane są wszystkie poprzednie ustawienia klasy, pozostawiając jedynie podstawową strukturę obiektu
typeof(*obiekt*)	Zwraca nazwę typu danych przechowywanych w obiekcie
mode(*obiekt*)	Zwraca informację o sposobie przechowywania obiektu w pamięci. W większości przypadków wynik będzie zgodny z **typeof()**, choć typy takie jak **integer** i **double** mają tę samą reprezentację wewnętrzną i zostaną zaklasyfikowane jako **numeric** – por. tabela 4
attributes(*obiekt*)	Zwraca listę wszystkich atrybutów obiektu. Funkcję można również wykorzystać do modyfikacji wartości poszczególnych atrybutów
attr(*obiekt*, *atrybut*)	Zwraca wartość konkretnego atrybutu obiektu; używając operatora <- można zmienić wartość tego atrybutu
comment(*obiekt*)	Pozwala na wyświetlenie lub zmianę komentarza przypisanego do obiektu, który nie jest wyświetlany automatycznie przez funkcję **print()**
str(*obiekt*)	Dostarcza zwięzłą informację o strukturze obiektu, w tym nazwach, typach i rozmiarach jego składowych
object.size(*obiekt*)	Zwraca liczbę bajtów pamięci zajmowanych przez obiekt

3.14. Struktura zbioru danych. Statystyki opisowe

Aby uzyskać informacje o strukturze wybranego zbioru danych można wykorzystać m.in. funkcje **str()** lub **glimpse()** z pakietu **dplyr**, które pozwalają w prosty sposób zapoznać się z podstawowymi właściwościami analizowanego zbioru. Bazowa funkcja **str()** zwraca informacje o typie obiektu, liczbie elementów, a w przypadku ramek danych – o liczbie wierszy i kolumn, typach poszczególnych zmiennych oraz przykładowych wartościach. Podobną funkcjonalność, lecz w nieco odmiennej formie, oferuje funkcja **glimpse()** dostępna w pakiecie **dplyr**.

Wyznaczanie statystyk opisowych (podsumowujących) stanowi jedno z podstawowych narzędzi analizy danych – polega ono na wykorzystaniu wybranych miar statystycznych do opisania właściwości cech. Pozwalają one na wstępne rozpoznanie charakterystyki badanych zmiennych, a także umożliwiają syntetyczne przedstawienie informacji o zbiorze danych; dzięki temu można m.in. ocenić ogólne właściwości badanej zmiennej (zmiennych) i wykryć potencjalne nieprawidłowości w danych. W R dostępnych jest wiele funkcji umożliwiających wyznaczenie najczęściej stosowanych statystyk zmiennych. Pewne funkcje indywidualne (zwracające pojedynczą wartość, np. średnia arytmetyczną – funkcja **mean()**) zostały

przedstawione już w tabeli 11. Należy jednakże pamiętać, że istnieją różne typy zmiennych: ilościowe (w języku R reprezentowane przez typ **numeric**) i jakościowe (typ **factor**). W tym miejscu Student powinien przypomnieć sobie stosowną klasyfikację, np. według Stanleya Stevensa. W języku R do wyznaczania statystyk opisowych dla zadanego zbioru danych można użyć funkcji takich jak m.in. (uwaga: wynik zawiera różne postacie statystyk opisowych dla cech kategorialnych i dla cech ciągłych):

- **summary()** – zwraca podstawowe charakterystyki zmiennej ilościowej: wartość minimalną, pierwszy kwartyl, medianę, średnią arytmetyczną, trzeci kwartyl oraz wartość maksymalną; dla zmiennych jakościowych przedstawiane są liczności poszczególnych kategorii,
- **describe()** z pakietu **psych** – dla zmiennych ilościowych zwraca m.in.: nazwę zmiennej, jej numer w zbiorze danych (**var**), liczbę wartości w zbiorze (**n**), średnią arytmetyczną(**mean**), odchylenie standardowe (**sd**), medianę (**median**), elementy minimalny (**min**) i element maksymalny (**max**), rozstęp (**range**), skośność (**skew**) oraz kurtozę (**kurtosis**),
- **describeBy(*nazwa_zbioru*, *group = 'zmienna*') z pakietu **psych** – wyznacza statystyki w podgrupach wyznaczonych ze względu na zmienną ***zmienna***.

W przypadku zmiennych jakościowych przydatne są także funkcje **table()** i **prop.table()**, które pozwalają obliczyć liczności oraz proporcje wystąpień poszczególnych kategorii.

Przykład 3.11.

Ramki danych – wybrane operacje. Struktura zbioru danych. Statystyki opisowe.

Uruchom program RStudio Desktop i wpisz poniższe polecenia w konsoli (bez komentarzy po znaku #). Po wprowadzeniu każdego z poleceń naciśnij klawisz ***Enter***.

```
studenci <- data.frame(           #ramka danych o studentach
  id = 1:6,
  imie = c("Anna", "Bartek", "Celina", "Daniel", "Ewa", "Filip"),
  wiek = c(22, 23, 21, 24, 21, 26),
  kierunek = c("ID", "INF", "ID", "MAT", "ID", "INF")
)
print(studenci)
str(studenci)     #informacje o strukturze ramki danych
studenci$srednia <- c(4.5, 3.8, 4.2, 4.9, 3.5, 4.2)     #nowa kolumna
print(studenci)
```

```
str(studenci)
typeof(studenci)
names(studenci)          #aktualne nazwy kolumn
row.names(studenci) <- c("S1","S2","S3","S4","S5","S6")    #nazwy wierszy
print(studenci)
subset(studenci, kierunek == "ID")    #studenci kierunku "ID"
studenci$imie                  #kolumna "imie"
studenci[1, ]                  #pierwszy wiersz
studenci[c(2,4), "srednia"]    #średnia ocen 2. i 4. studenta
studenci[studenci$srednia > 4, ]   #tylko studenci ze średnią ocen > 4
studenci[studenci$wiek < 23, ]     #tylko studenci w wieku < 23 lata
      #sortowanie po średniej ocen (malejąco)
studenci_sort <- studenci[order(studenci$srednia, decreasing = TRUE), ]
print(studenci_sort)
class(studenci)             #data.frame
typeof(studenci)            #list
object.size(studenci)
      #średnia ocen dla każdego kierunku studiów
aggregate(srednia ~ kierunek, data = studenci, FUN = mean)
with(studenci, mean(wiek))     # średni wiek studentów
nowy <- data.frame(       #nowy student
  id = 7, imie = "Grzegorz", wiek = 25, kierunek = "MAT", srednia = 4.0)
studenci <- rbind(studenci, nowy)     #dodanie nowego wiersza
print(studenci)
      #nowa kolumna "wiek2" – wiek podniesiony do kwadratu
studenci <- within(studenci, {
  wiek2 <- wiek^2
})
print(studenci)
summary(studenci)           #podstawowe statystyki opisowe, sposób nr 1
table(studenci$kierunek)              #liczności dla zmiennej jakościowej
prop.table(table(studenci$kierunek))  #rozkłady procentowe
library(psych)
describe(studenci$wiek)     #podstawowe statystyki opisowe, sposób nr 2
describeBy(studenci, group = "kierunek")     #statystyki w podgrupach (ze
                                             #względu na kierunek)
```

4. Modyfikacja przepływu sterowania

W niniejszej części omówione zostaną zarówno podstawowe instrukcje warunkowe (pozwalające wykonywać różne fragmenty kodu w zależności od spełnienia określonego warunku), jak również wybrane pętle iteracyjne (umożliwiające wielokrotne wykonywanie fragmentów kodu).

Przy tworzeniu niniejszego rozdziału korzystano z pomocy programu R, dokumentacji R ("Rdocumentation", 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN ("The Comprehensive R Archive Network", 2025) i jego podstron (gdzie znajdują się m.in. opisy poszczególnych pakietów i funkcji) oraz pozycji literaturowych (Biecek, 2018; Gągolewski, 2014; Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012).

4.1. Instrukcja warunkowa if

Niejednokrotnie zachodzi potrzeba sprawdzenia jakiegoś warunku. W zależności od tego, czy jest on prawdziwy bądź fałszywy, wykonane mogą zostać różne fragmenty kodu. W R najprostsza postać instrukcji warunkowej to: **if (*warunek*) *instrukcja*** (w przypadku wykonania tylko i wyłącznie 1 instrukcji) lub nieco bardziej rozbudowana postać:

```
if (warunek) {
        #instrukcje do wykonania
}
```

Instrukcja warunkowa **if** jest często łączona do postaci instrukcji **if…else**, gdzie po przetestowaniu pierwszego warunku logicznego i otrzymaniu wyniku **FALSE** wykonywane są instrukcje alternatywne zapisane w bloku **else**:

```
if (warunek) {
        #instrukcje do wykonania
} else {
        #instrukcje do wykonania
}
```

W sytuacji potrzeby rozbudowania „drzewa decyzyjnego", można przetestować więcej wyrażeń logicznych w blokach **if…else if…else**. Jeśli żadne z wyrażeń w blokach **if** bądź **else if** nie jest prawdziwe, wtedy wykonuje się fragment kodu zapisany w bloku **else**. Ogólna postać instrukcji warunkowej **if** (jeżeli) w języku R to:

Politechnika Świętokrzyska Kielce University of Technology
Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki" nr FERS.01.05-IP.08-0234/23

```
if (warunek1) {
        #instrukcje do wykonania
} else if (warunek2) {
        #instrukcje do wykonania
} else {
        #instrukcje do wykonania
}
```

Warunkiem testowym może być tylko i wyłącznie jednoelementowy wektor logiczny zawierający jako element wartość **TRUE** lub **FALSE**. Operatory relacyjne i logiczne, które są wykorzystywane przy przybudowie wyrażeń logicznych niezbędnych do tworzenia instrukcji warunkowej **if** zostały omówione w podrozdziale 3.2.

Przykład 4.1.

Instrukcja warunkowa if.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **jezeli.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
      #pobranie wartości liczbowej od użytkownika
x <- as.numeric(readline(prompt="Podaj jedną liczbę całkowitą x: "))

#pierwsza instrukcja warunkowa
if (x > 10) {
  print("x jest większe niż 10")
} else if (x == 10) {
  print("x jest równe 10")
} else {
  print("x jest mniejsze niż 10")
}

      #czy liczba x jest parzysta?
if (x %% 2 == 0) {
  print("Liczba x jest parzysta")
} else {
  print("Liczba x jest nieparzysta ")
}
```

4.2. Funkcja ifelse()

W języku R, oprócz klasycznej instrukcji warunkowej **if**, dostępna jest także funkcja **ifelse()**, która w wielu sytuacjach zastępuje pętle i pozwala tworzyć krótszy, zwektoryzowany kod. Jest ona szczególnie przydatna, gdy zaistnieje potrzeba zastosowania określonego warunku na całym wektorze danych – działa ona bowiem element po elemencie. Jej składnia to:
ifelse(*warunek, wartosc_prawda, wartosc_fałsz*)
gdzie ***warunek*** to warunek logiczny (może też być wektorem wartości logicznych); ***wartosc_prawda*** – wartość zwracana, gdy warunek jest spełniony; ***wartosc_fałsz*** – wartość zwracana, gdy warunek nie jest spełniony.

Funkcja **ifelse() z**awsze zwraca wektor o tej samej długości co ***warunek***.
W wynikowym wektorze będzie znajdowała się wartość ***wartosc_prawda*** dla tych elementów, gdzie warunek jest prawdziwy, a w przeciwnym wypadku – wartość ***wartosc_fałsz***.

Przykład 4.2.

Funkcja ifelse().

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **jezeli.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
x <- c(-2, -1, 0, 1, 2, 3, 0)
ifelse(x >= 0, "nieujemna", "ujemna")    #czy liczba jest ujemna?
x1 <- ifelse(x == 0, NA, x)        #zastępowanie wartości 0 wartościami NA
print(x1)

      #oceny Studentów
oceny <- c(95, 76, 58, 40, 83)
kategorie <- ifelse(oceny >= 90, "bardzo dobry",
             ifelse(oceny >= 70, "dobry",
             ifelse(oceny >= 50, "dostateczny", "niedostateczny")))
print(kategorie)
```

4.3. Instrukcja switch

W sytuacji wielu przypadków do sprawdzenia, a także wielu możliwych akcji do wykonania, użyteczne może być wykorzystanie funkcji **switch()**:

switch (*x*,
***wartość1* = *wynik1*,**
***wartość2* = *wynik2*,**
***wynik_domyślny*)**

Pierwszy argument ***x*** to zmienna, której wartość jest testowana. Kolejne argumenty to wartości, z którymi po kolei jest porównywalna zmienna ***x***, a także wynik zwracany w przypadku ewentualnego pozytywnego wyniku porównania. Gdy pierwszy argument jest typu **character**, to następuje dopasowanie do nazwy argumentu, a jeśli wartością numeryczną – wybierany jest argument o danym numerze pozycji. Ostatni argument to ***wynik_domyślny*** – jest zwracany w sytuacji, gdy żadne z powyższych porównań nie jest zakończone powodzeniem (akcja domyślna).

Przykład 4.3.

Instrukcja switch.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **switch.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

Uruchom kod 7 razy, zmieniając każdorazowo nazwę dnia tygodnia (zmienna **dzień**).

```
dzien <- "wtorek"
wynik <- switch(dzien,
  "poniedziałek" = "Początek tygodnia",
  "piątek" = "Prawie weekend",
  "sobota" = "Weekend!",
  "niedziela" = "Odpoczynek",
  "Dzień roboczy"
)
print(wynik)
```

4.4. Pętla iteracyjna for

Pętle iteracyjne pozwalają na cykliczne wykonywanie pewnych instrukcji. Pętlę **for** wykorzystuje się zazwyczaj, gdy liczba wykonywanych operacji jest znana (w tym np. do „przechodzenia” po elementach wektora, listy, macierzy, tablicy itd., choć w przypadku struktur wielowymiarowych czasami przydatne są też pętle zagnieżdżone).

Pętla **for** jest deklarowana słowem kluczowym **for**, do którego przekazywany jest trzyczęściowy argument. Pierwsza część to nazwa zmiennej, do której iteracyjnie przypisywane są wartości kolejnych elementów wektora z części trzeciej (którą może stanowić dowolny wektor z wartościami dowolnego typu). Druga część to słowo kluczowe **in**. Ogólna postać instrukcji **for**:

for(licznik in początek:koniec) **{**
    #instrukcje do wykonania
}

W ogólności pętla for wykonuje się **length(*wektor*)** razy, a więc dla każdego elementu wektora stanowiącego trzecią część argumentu:

for(licznik in wektor) **{**
    #instrukcje do wykonania
}

W pętli for można wykorzystać jedną z dwóch instrukcji: **break** służy do zatrzymania przebiegu pętli, natomiast **next** do przejścia do kolejnej iteracji. Jeśli pętla iteracyjna jest umieszczona wewnątrz funkcji, to w każdej chwili można ją zakończyć (tak jak i wykonanie całej funkcji) poleceniem **return()**.

Przykład 4.4.

Pętla iteracyjna for.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **for.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
wektor <- c(2, 4, 6, 8, 10)
     #dla każdego element wektora wartość podniesiona do kwadratu
for (elem in wektor) {
  print(paste("Kwadrat liczby", elem, "wynosi", elem^2))
}

     #instrukcje next I break
for (i in 1:10) {
  if (i == 5) next      #pomiń liczbę 5
  if (i > 8) break      #przerwij pętlę po 8
  print(i)
}
```

4.5. Pętla iteracyjna while

Działanie pętli **while** polega na cyklicznym wykonywaniu pewnych instrukcji, dopóki spełniony jest określony warunek. Wykorzystuje się ją zazwyczaj, gdy liczba wykonywanych operacji nie jest znana. Warunek jest podawany przed blokiem instrukcji i jest testowany przed każdym wykonaniem pętli. Ogólna postać instrukcji **while**:

```
while(warunek) {
        #instrukcje do wykonania
}
```

Gdzieś w pętli musi być umieszczony modyfikator warunku, by nie wykonywała się ona w nieskończoność. Można także wykorzystać instrukcję **break**, która pozwala na przerwanie wykonywania całej pętli w dowolnym momencie. Wywołanie instrukcji **next** umożliwia z kolei natychmiastowe zakończenie bieżącej iteracji pętli i przejście do kolejnej iteracji, która zawsze rozpoczyna się od sprawdzenia warunku testowego.

Przykład 4.5.

Pętla iteracyjna while.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **while.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
licznik <- 1
while (licznik <= 5) {
  print(paste("Iteracja nr", licznik))
  licznik <- licznik + 1
}
```

Przykład 4.6.

Utwórz program, który wczytuje liczby od użytkownika, aż do podania liczby ujemnej, a następnie oblicza sumę i średnią arytmetyczną podanych wartości (bez tej ujemnej).

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **liczby_while.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchomić w całości.

```
suma <- 0                   #zmienna do sumowania
liczba <- 0                 #zmienna pomocnicza
licznik <- 0                #ile liczb zostało wprowadzonych

while (liczba >= 0) {
  liczba <- as.numeric(readline(prompt = "Podaj liczbę: "))
  if (liczba >= 0) {
    suma <- suma + liczba
    licznik <- licznik + 1
  }
}
if (licznik > 0) {
  srednia <- suma / licznik
  cat("Suma podanych liczb wynosi:", suma, "\n")
  cat("Średnia arytmetyczna wynosi:", srednia)
} else {
  cat("Nie podano żadnej liczby nieujemnej!")
}
```

4.6. Pętla iteracyjna repeat

W R pętla **repeat** działa podobnie jak pętla **while**, w której warunek testowy ustawiony jest zawsze na **TRUE**. Wykonywanie pętli trwa więc w nieskończoność, dopóki nie zostanie jawnie przerwane. Należy więc zawsze wywołać instrukcję **break** (ew. **return()** wewnątrz funkcji), by uzyskać pożądany wynik i zatrzymać działanie pętli w odpowiedniej chwili. Ogólna postać pętli **repeat**:

```
repeat {
    #instrukcje do wykonania
    If(!warunek) break
}
```

W przeciwieństwie do pętli **while** następuje co najmniej jedna iteracja tej pętli.

Przykład 4.7.

Pętla iteracyjna repeat.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **repeat.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
licznik2 <- 1
repeat {
  print(paste("Iteracja repeat:", licznik2))
  licznik2 <- licznik2 + 1
  if (licznik2 > 3) break    #zatrzymanie po 3 iteracjach
}
```

5. R jako język funkcyjny

R jest językiem funkcyjnym, co oznacza, że każde działanie sprowadza się do wywołania jakiejś funkcji. Dotyczy to również operatorów, które są szczególnym rodzajem funkcji wbudowanych, np. wyrażenie **Y + Z** odpowiada wywołaniu **"+"(Y, Z)** – tj. funkcji o nazwie „**+**" operującej na argumentach **Y** oraz **Z**.

Przy tworzeniu niniejszego rozdziału korzystano z pomocy programu R, dokumentacji R ("Rdocumentation", 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN ("The Comprehensive R Archive Network", 2025) i jego podstron (gdzie znajdują się m.in. opisy poszczególnych pakietów i funkcji) oraz pozycji literaturowych (Biecek, 2018; Gągolewski, 2014; Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012).

5.1. Funkcje w R

Funkcje są szczególnie przydatne, gdy dany fragment kodu ma być wykorzystywany wielokrotnie, często dla odmiennych danych, w różnych miejscach programu. Pojęcie **podprogramu** powinno być już znane Studentowi – w przypadku wątpliwości należy sięgnąć do dostępnych materiałów. Wiele wbudowanych funkcji zostało już przedstawionych w niniejszym opracowaniu.

Funkcje można traktować jako swoistą „skrzynkę": na wejściu dostarczane są różne obiekty (argumenty), a na wyjściu zwracany jest wynik obliczeń. Ponieważ w R funkcje są obiektami, mogą być przypisywane do zmiennych, czy też przekazywane innym funkcjom jako argumenty. Do definiowania funkcji należy wykorzystać wyrażenie:

```
function(lista_argumentów) {
    #ciało_funkcji
}
```

gdzie: ***ciało_funkcji*** to operacje/ wyrażenia do wykonania na obiektach określonych przez ***listę_argumentów***.

Aby móc korzystać z danej funkcji w różnych miejscach programu, musi być ona powiązana z określoną nazwą, pod którą będzie później wywoływana. Jak wspomniano powyżej, funkcje można przypisywać (np. przy wykorzystaniu operatora **<-**) do obiektów tak samo, jak dowolne inne zmienne.

Lista argumentów (która występuje po słowie kluczowym **function**) może być pusta (gdy funkcja nie potrzebuje danych wejściowych) lub też może zawierać dowolną liczbę argumentów. W ogólności, ***lista_argumentów*** składa się z par, które mogą, ale nie muszą występować – każda para obejmuje nazwę parametru (pod taką nazwą będzie dostępny w funkcji obiekt przekazany przy wywołaniu) oraz ewentualnie jego wartość domyślną. Argumenty rozdziela się przecinkami. Wewnątrz ***ciała funkcji*** zapisywane są instrukcje, które mają zostać wykonane; ten zasadniczy kod najczęściej umieszcza się wewnątrz nawiasów klamrowych – chyba, że funkcja zawiera tylko jeden wiersz kodu. Wynikiem funkcji jest wartość obliczona w ostatniej linii lub jawnie zwrócona za pomocą instrukcji **return()** – w ogólności wywołanie tego polecenia powoduje przerwanie wykonywania funkcji, a także przekazanie jako wyniku wartości będącej argumentem polecenia **return()**.

Warto pamiętać, że argumenty przekazywane są przez wartość, czyli funkcja otrzymuje kopię obiektów i nie modyfikuje ich oryginałów. Dla czytelności kodu zaleca się, aby kod umieszczony w ciele funkcji wyróżniać odpowiednimi wcięciami. Argumenty funkcji oraz obiekty nazwane tworzone wewnątrz funkcji mają **zasięg lokalny**, co oznacza, że po zakończeniu jej działania nie są już dostępne na zewnątrz. Dobrą praktyką jest ograniczanie odwołań do obiektów globalnych i korzystanie przede wszystkim z danych jawnie przekazanych funkcji jako argumenty. Parametr funkcji to nazwa umieszczona w jej definicji na liście parametrów, natomiast argument jest konkretną wartością przypisywaną temu parametrowi przy wywołaniu funkcji. Przykładowo, można zdefiniować funkcję z parametrami **A** i **B**, a następnie wywołać ją z argumentami **A = 10** i **B = 5**.

Istotną cechą R jest tzw. **zasada „leniwej ewaluacji”** (ang. *lazy evaluation*) – wyrażenia przekazane do funkcji są obliczane dopiero wtedy, gdy faktycznie są potrzebne, gdyż nie zawsze funkcja musi korzystać ze wszystkich przekazanych jej argumentów. Dodatkowo, na liście argumentów może pojawić się parametr specjalny **…**, co pozwala na przekazanie dowolnej liczby argumentów.

Jeśli funkcja jest uruchamiana bezpośrednio w konsoli, nie zawsze konieczne jest wyświetlanie jej wyniku. W takich przypadkach można posłużyć się funkcją **invisible()**, która „ukrywa” zwracaną wartość. Funkcje w R nie muszą mieć jednakże przypisanej nazwy – wówczas nazywa się je **funkcjami anonimowymi**.

Przy definiowaniu nowych funkcji warto jasno określić, jakie zadanie mają dokładnie realizować oraz jakie dane wejściowe będą dla nich właściwe. Istotne jest również rozróżnienie, czy funkcja powinna działać na całych wektorach, czy tylko na pojedynczych wartościach. W przypadku funkcji operujących na wektorach należy

upewnić się, że poprawnie radzą sobie zarówno z pustymi wektorami, jak i z wektorami jedno- oraz wieloelementowymi.

Funkcje w R są więc podstawowym narzędziem umożliwiającym pisanie czytelnego kodu. Dzięki nim można działać na różnym poziomie abstrakcji – raz przygotowaną funkcję można następnie wykonywać wielokrotnie dla różnych danych.

Przykład 5.1.

Funkcje w języku R.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **funkcje.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
     #prosta funkcja – suma kwadratów dwóch liczb
suma_kwadratow <- function(a, b) {
  wynik <- a^2 + b^2
  return(wynik)
}
print(suma_kwadratow(3, 4))    #wynik: 25

     #funkcja z wartością domyślną dla argumentu
potegowanie <- function(x, n = 2) {
  return(x^n)
}
potegowanie(6)       #wynik: 36 (n = 2 domyślnie)
potegowanie(2, 3)    #wynik: 8

(function(x) x + 10)(7)   #funkcja anonimowa; wynik: 17

     #funkcja przyjmująca dowolną liczbę argumentów
suma_wielu <- function(...) {
  return(sum(...))
}
suma_wielu(1, 2, 3, 4, 5)    #wynik: 15
```

5.2. Sprawdzanie poprawności danych

Walidacja danych jest kluczowym elementem zapewniającym poprawność działania funkcji i bezpieczeństwo obliczeń. R nie ma możliwości powstrzymania użytkownika od wywołania funkcji na niewłaściwych danych. Przed przystąpieniem do właściwych

operacji, zaleca się sprawdzenie, czy dane wejściowe spełniają wymagania funkcji pod względem typu, długości, zakresu wartości oraz struktury. Można m.in. sprawdzić, czy argument jest wartością liczbową (wektorem wartości liczbowych) o np. elementach dodatnich, całkowitych itd. W tym celu najczęściej stosuje się funkcje wbudowane, takie jak np.: **is.numeric()**, **is.character()**, **is.logical()**, a także funkcje sprawdzające właściwości wektorów, macierzy czy ramek danych, np. **length()**.

Pomocne może być wykorzystanie funkcji **stopifnot()**, która weryfikuje, czy wszystkie przekazane jej warunki są prawdziwe (**TRUE**), a jeżeli są spełnione, funkcja kontynuuje swoje działanie. W przeciwnym wypadku (tj. gdy którekolwiek z wyrażeń nie jest prawdziwe), automatycznie wywoływana jest funkcja **stop()**, która przerywa działanie funkcji i wyświetla komunikat o pierwszym napotkanym błędzie. Ostrzeżenia generuje funkcja **warning()**. Aby wyłączyć (globalnie) komunikaty ostrzegawcze, należy zmienić wartości opcji ***warn*** i nadać jej wartość -1: **options(*warn* = *-1*)**. Należy jednak ostrożnie stosować tą własność. Wartość domyślna to 1: **options(*warn* = *1*)**. Do odczytu aktualnej wartości ***warn*** służy polecenie: **getOption("*warn*").**

Funkcja **tryCatch()** służy do przechwytywania i obsługi błędów (argument ***error***) i ostrzeżeń (argument ***warning***; mniej poważnych od błędu, które nie przerywają wykonywania kodu) oraz pozwala wyświetlić niestandardowe komunikaty. Stara się ona uruchomić fragment danego kodu, a gdy pojawi się błąd, to wykonuje alternatywne obliczenia. Można także dodać (argument ***finally***) blok kodu, który zostanie wykonany zawsze – niezależnie od tego, czy wystąpił błąd lub ostrzeżenie. Składnia funkcji:

```
tryCatch({
        #kod do uruchomienia
},  error = function(e) {
        #kod wykonywany w przypadku wystąpienia błędu
  },
   warning = function(w) {
        #kod wykonywany w przypadku wystąpienia ostrzeżenia
} , finally = {
         #kod wykonywany zawsze
})
```

Oprócz powyższych, funkcja **tryCatch()** udostępnia również bardziej zaawansowane mechanizmy scenariuszy obsługi błędów (np. obsługa określonych typów błędów).

Przykład 5.2.

Wybrane aspekty sprawdzania poprawności danych w języku R.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **bledy.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
srednia2 <- function(x) {
      #sprawdzenie: czy wektor jest liczbowy (bez wartości NA) i niepusty
  stopifnot("Niepoprawne dane!" = !is.na(as.numeric(x)), length(x) > 0)
  return(mean(x))
  }
srednia2(c(2, 4, 6, 8))      #poprawne dane
srednia2(c(1, 2, NA, 4))     #błąd
srednia2("tekst")            #błąd

      #instrukcja tryCatch()
pierwiastek <- function(x) {
  tryCatch({
    print(sqrt(x))             #pierwiastek kwadratowy
  },  error = function(e) {
    print("błąd")
  },
  warning = function(w) {
    print("ostrzeżenie")
  } , finally = {
    print("Kod wykonywany zawsze!")
  })
}
pierwiastek(9)      #wynik: 3; brak błędu lub ostrzeżenia
pierwiastek("ID")   # próba pierwiastkowania tekstu – błąd
pierwiastek(-1)     #pierwiastek kwadratowy z liczby ujemnej - ostrzeżenie
pierwiastek(as.numeric(readline("Podaj liczbę: ")))
```

6. Podstawowe aspekty przetwarzania plików w R

Aby możliwe było prowadzenie obliczeń, konieczne jest zapewnienie dostępu do odpowiednich danych. W praktyce, często zbiór danych zapisany jest w postaci pliku na dysku twardym lub znajduje się w zasobach sieciowych. Zbiór taki trzeba najpierw wczytać i przekształcić do postaci obiektu rozpoznawalnego przez język R, do czego należy wykorzystać pewne narzędzia. Zgromadzone dane mogą przyjmować różne formaty, a ich poprawne odczytanie i przygotowanie stanowi pierwszy etap procesu analizy.

Przy tworzeniu niniejszego rozdziału korzystano z pomocy programu R, dokumentacji R (“Rdocumentation“, 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN (“The Comprehensive R Archive Network“, 2025) i jego podstron (gdzie znajdują się m.in. opisy poszczególnych pakietów i funkcji) oraz pozycji literaturowych (Biecek, 2018; Gągolewski, 2014; Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012), a także (“Scraping Data“, 2025.)

6.1. Podstawowe operacje na plikach i katalogach

Podczas pracy z plikami należy zawsze dokładnie wskazać obiekt znajdujący się na dysku. W tym celu wykorzystuje się tzw. **ścieżkę dostępu** (ang. *path*), która obejmuje zarówno nazwę pliku, jak i jego położenie w hierarchii katalogów. Należy pamiętać, że sposób zapisu ścieżek zależy od używanego systemu operacyjnego: w systemach **UNIX/Linux** stosuje się slash (ukośnik) „**/**”, natomiast w systemach **Windows** znak odwrotnego ukośnika (backslash) „****”. W R dostępna jest funkcja **normalizePath()**, która pozwala na przekształcenie ścieżki do poprawnego formatu dla danego systemu operacyjnego, a także pozwala sprawdzić, czy wskazany plik faktycznie istnieje na dysku.

W podrozdziale 1.3.3. objaśniono mechanizm przechowywanie danych przez RStudio oraz pojęcie katalogu roboczego.

Z poziomu języka R można zarządzań plikami i folderami, które znajdują się na dysku. Wybrane funkcje do wykonywania podstawowych operacji:

- **file.exists()** lub **dir.exists()** – sprawdza, czy dany plik lub katalog istnieje,
- **file.info()** – zwraca szczegółowe informacje o plikach lub katalogach we wskazanej ścieżce dostępu w formacie ramki danych,
- **list.dirs()** – zwraca nazwy katalogów znajdujących się we wskazanej ścieżce,

- **dir()** lub **list.files()** – domyślnie zwracają nazwy wszystkich plików i katalogów znajdujących się w podanej ścieżce (ewentualnie można także sprawdzać wszystkie podkatalogi – argument ***recursive=TRUE***),
- **file.create()** oraz **dir.create()** – tworzenie nowych plików lub katalogów; jeśli plik o danej ścieżce istnieje, jego zawartość zostanie wymazana!
- **file.remove()** – usuwanie plików (w niektórych systemach operacyjnych również katalogów),
- **unlink()** – usuwanie katalogów (wraz z zawartością),
- **file.copy()** – kopiowanie plików bądź katalogów (wraz z zawartością),
- **file.rename()** – zmiana nazwy obiektu.

Przykład 6.1.

Podstawowe operacje na plikach i katalogach w R.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **pliki_katalogi.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
R.home()       #ścieżka katalogu, w którym zainstalowany jest R
tempdir()      #ścieżka do katalogu, w którym bieżąca sesja R przechowuje
               #dane tymczasowe
getwd()        #sprawdzenie adresu katalogu roboczego
setwd(" ")  #tu należy ustawić właściwy katalog roboczy!! np. Ctrl+Shift+H
list.files()   #lista wszystkich plików w katalogu roboczym
file.exists("dane.txt")      #czy w kat. roboczym istnieje plik dane.txt?
dir.exists("f1")             #czy w kat. roboczym istnieje folder f1?
file.create("dane.txt")      #utworzenie pliku
dir.create("f1")             #utworzenie katalogu
file.exists("dane.txt")      #czy w kat. roboczym istnieje plik dane.txt?
dir.exists("f1")             #czy w kat. roboczym istnieje folder f1?
file.info("dane.txt")        #informacje o pliku
list.files()                 #lista wszystkich plików w katalogu roboczym
file.rename("dane.txt", "nazwa2.txt")    #zmiana nazwy pliku
file.copy("nazwa2.txt", "nazwa2_kopia.txt")
list.files()                 #lista wszystkich plików w katalogu roboczym
file.remove("nazwa2_kopia.txt")          #usunięcie pliku
list.files()                 #lista wszystkich plików w katalogu roboczym
```

6.2. Wczytywanie i zapis danych do plików tekstowych

Pliki tekstowe stanowią jedną z najprostszych i jednocześnie najczęściej spotykanych form przechowywania informacji. W praktyce, za plik tekstowy uznaje się taki plik, którego zawartość można otworzyć i odczytać w zwykłym edytorze tekstowym, np. w Notatniku. Do tej kategorii zalicza się m.in.:

- kody źródłowe programów,
- strony internetowe HTML,
- pliki w formacie XML.

Pliki tekstowe mogą być postrzegane jako **wektory napisów** (gdy każdy napis to oddzielny wiersz kodu) lub jako **pojedynczy napis** (gdy zawartość całego pliku traktowana jest jako jedna całość).

Każdy plik tekstowy przechowuje znaki w określonym systemie kodowania (np. UTF-8, ASCII). Z tego względu podczas wczytywania i zapisywania danych należy zwrócić uwagę na poprawne dopasowanie kodowania, aby uniknąć problemów z „błędnymi" znakami. Dodatkowo, systemy operacyjne różnią się sposobem oznaczania końca linii (**EOL**, ang. *end-of-line markers*):

- w systemach **Windows** stosuje się sekwencję CRLF (\r \n),
- w systemach **Unix/Linux** – znak LF (\n),
- w starszych wersjach **MacOS** – znak CR (\r).

W analizie danych szczególne znaczenie mają pliki tekstowe o poniższych rozszerzeniach, które pełnią funkcję standardowych formatów wymiany danych:

- **.txt** (ang. *text file*) – zwykły plik tekstowy,
- **.csv** (ang. *comma separated values*) – wartości są oddzielone przecinkiem,
- **.tsv** (ang. *tab separated values*) – wartości oddzielane są znakiem tabulacji.

W powyższych plikach kolejne wartości kolumn są rozdzielane za pomocą różnego znaku (separatora).

Aby w R wczytać dane tekstowe do ramki danych, można wykorzystać funkcję **read.table()** z pakietu **utils**. Wybrane, często używane argumenty:

- ***file = "…/plik.csv"*** – ścieżka do pliku tekstowego,
- ***sep = ";"*** – symbol używany jako separator pól; w tym przypadku jest to średnik,
- ***dec = ","*** – znak pełniący funkcję separatora części dziesiętnej (tutaj przecinek),

- ***header = TRUE/ FALSE*** – pierwszy wiersz zawiera (**TRUE**) lub nie (**FALSE**) nazwy kolumn,
- ***fileEncoding = "UTF-8"*** – kodowanie znaków (tutaj UTF-8).

Za pomocą operatora <- można wczytać dane do zmiennej, co pozwala na dalszą pracę z wczytanym zbiorem.

W pakiecie **utils** znajdują się również cztery inne funkcje o podobnym działaniu jak **read.table()**, ale odmiennych wartościach domyślnych:

- **read.csv(*plik, header=TRUE, sep=",", quote="\"", dec=".", fill=TRUE, comment.char="", ...*)**,
- **read.csv2(*plik, header=TRUE, sep=";", quote="\"", dec=",", fill=TRUE, comment.char="", ...*)**,
- **read.delim(*plik, header=TRUE, sep="\t", quote="\"", dec=".", fill=TRUE, comment.char="", ...*)**,
- **read.delim2(*plik, header=TRUE, sep="\t", quote="\"", dec=",", fill=TRUE, comment.char="", ...*)**.

Należy wyraźnie zaznaczyć, że do wczytywania danych tekstowych w języku R do struktury **tibble** (por. podrozdział 7.2) bardzo często wykorzystuje się pakiet **readr**, gdzie najbardziej ogólną funkcje stanowi **read_delim()** – warto zapoznać się z materiałami o tym pakiecie: https://readr.tidyverse.org/.

Do wczytywanie danych z dużych plików warto rozważyć wykorzystanie funkcji **data.table::fread()**.

Do wczytania danych, które nie muszą mieć formy regularnej tabeli**,** można wykorzystać funkcję **scan()**. Poszczególne wiersze mogą zawierać różną liczbę elementów. Jej wynikiem jest wektor odczytanych wartości, którego typ zależy od ustawionych parametrów: **wektor <- scan(*"nazwa.pliku"*)**. Najważniejsze argumenty:

- ***file*** – ścieżka do pliku tekstowego,
- ***what*** – określa typ danych do wczytania (np. ***logical***, ***integer***, ***character***),
- ***sep*** – określa separator dla kolejnych wartości,
- ***nmax*** – umożliwia określenie maksymalnej liczby wartości do odczytania.

Kolejna funkcja pozwalająca na odczyt danych to **readLines()**, której zadaniem jest odczytanie wskazanej liczby linii tekstu z pliku lub innego źródła traktowanego jako tzw. połączenie. Domyślnie odczytywane są wszystkie linie. Rezultatem działania

funkcji jest **wektor napisów**. Dodatkowymi argumentami można określić m.in. sposób kodowania znaków. Domyślnie, jeżeli na końcu pliku nie ma pustego wiersza, funkcja **readLines()** zgłosi ostrzeżenie.

Aby zapisać dane obiektów R (np. wartość zmiennej) do pliku tekstowego w określonym formacie, zastosowana może zostać funkcja **write.table()** z pakietu **utils**. Wybrane argumenty:

- ***x*** – obiekt do zapisania,
- ***file*** – ścieżka do pliku, do którego dane zostaną zapisane,
- ***append*** – określa, czy dopisywać dane do istniejącego pliku (**TRUE**), czy nadpisać go (**FALSE**),
- ***quote*** – wskazuje, czy napisy mają być zapisywane w cudzysłowie (dla wartości **TRUE** będą),
- ***sep*** – określa separator dla kolejnych wartości,
- ***eol*** – znak końca wiersza,
- ***dec*** – znak używany jako separator dziesiętny,
- ***row.names*** i ***col.names*** – wskazują, czy zapisywać nazwy wierszy i kolumn.

Aby zdefiniować kodowanie UTF-8, należy zastosować argument: ***fileEncoding = "UTF-8"***.

Dodatkowo, w pakiecie **utils** dostępne są dwie funkcje o analogicznym działaniu (zapis w formacie *.csv), lecz określnych parametrach domyślnych:

- **write.csv(*x, file = "", sep = ",", dec=".", ...*)**,
- **write.csv2(*x, file = "", sep = ";", dec=",", ...*)**.

R udostępnia także kilka funkcji, które pozwalają zapisywać kolejne wiersze tekstu do pliku tekstowego – są to m.in.:

- **writeLines()** – zapisuje wiersze tekstu do „połączenia”, którym może być np. plik na dysku,
- **cat()** – oprócz wyświetlania wyniku w konsoli, pozwala na zapis do pliku (wykorzystując argument ***file***),
- **sink() –** przechwytuje całość wyjścia z konsoli R i zapisuje ją do wskazanego pliku; pozwala więc na zapis przebiegu interakcji z R (łącznie z wykonanymi poleceniami i otrzymanymi wynikami) do wskazanego pliku tekstowego zamiast na ekran – do momentu odwołania.

Przykład 6.2.

Wczytywanie i zapis danych do plików tekstowych.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **pliki_tekstowe.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
dane <- data.frame(        #utworzenie prostej ramki danych
  Imie = c("Anna", "Tomasz", "Kasia"),
  Wiek = c(23, 30, 27)
)
      #zapis danych z ramki do pliku CSV
write.csv(dane, file = "dane.csv", row.names = FALSE)
      #wczytanie danych z pliku CSV – zły separator!!
dane_wczytane <- read.table("dane.csv", sep = ";", header = TRUE)
print(dane_wczytane)
      #wczytanie danych z pliku CSV
dane_wczytane2 <- read.table("dane.csv", sep = ",", header = TRUE)
print(dane_wczytane2)
      #użycie funkcji ReadLines()
dane_wczytane3 <- readLines("dane.csv")
print(dane_wczytane3)

      #writeLines() i readLines()
writeLines(c("linia 1", "linia 2", "linia 3"), "tekst.txt")
readLines("tekst.txt")

sink(file = "historia.txt")   #przekierowanie wyjścia z konsoli R do pliku
print(dane_wczytane2)
print(str(dane_wczytane2))
sink()          #odwołanie przekierowania do pliku; sprawdź jego zawartość
```

6.3. Wczytywanie i zapis danych do arkuszy kalkulacyjnych

Dane w formie tabelarycznej są bardzo często przechowywane w arkuszu kalkulacyjnym (np. Microsoft Excel). Pliki tego typu posiadają zazwyczaj rozszerzenie **.xlsx** (nowszy format) lub **.xls** (starszy format). Dane takie można wczytać bezpośrednio do programu R z wykorzystaniem jednego z wielu dostępnych pakietów (np. **openxlsx**, **readxl**). W przypadku pakietu o nazwie **openxlsx**, niezbędna funkcja to **read.xlsx()**. Wśród licznych argumentów tej funkcji, szczególnie istotne są:

- ***xlsxFile*** – ścieżka do pliku xlsx, z którego mają zostać pobrane dane,
- ***sheet*** – numer lub nazwa arkusza w skoroszycie, z którego mają zostać odczytane dane,
- ***startRow*** – numer pierwszego wiersza, od którego ma się rozpocząć wczytywanie danych,
- ***colNames*** – określa, czy pierwszy wiersz danych zawiera nazwy kolumn (jeśli przyjmuje wartość **TRUE** – zawiera),
- ***rowNames*** – określa, czy pierwsza kolumna powinna zostać potraktowana jako nazwy wierszy (jeśli przyjmuje wartość **TRUE** – tak),
- ***skipEmptyRows*** oraz ***skipEmptyCols*** – pozwalają kontrolować, czy puste wiersze i kolumny mają być pomijane podczas wczytywania (jeśli przyjmuje wartość **TRUE** – będą pomijane),
- ***na.strings*** – określa, jakie wartości powinny być traktowane jako brakujące dane (**NA**).

Dane przechowywane w obiektach języka R (np. wektorach, macierzach czy ramkach danych) mogą być zapisywane bezpośrednio do plików w formacie arkuszy kalkulacyjnych **Excel**. W pakiecie **openxlsx** służy do tego funkcja **write.xlsx()**, która umożliwia szybki eksport danych do pliku z rozszerzeniem **.xlsx**. Funkcja ta pozwala nie tylko na zapis danych w standardowej postaci, ale także na kontrolę nad formatem zapisywanego pliku. Wśród argumentów funkcji **write.xlsx()** szczególnie istotne są:

- ***x*** – obiekt R (np. ramka danych), który ma zostać zapisany do pliku,
- ***file*** – pełna ścieżka i nazwa pliku xlsx, do którego zostaną zapisane dane,
- ***sheetName*** – nazwa arkusza, w którym zostaną zapisane dane,
- ***colNames*** – określa, czy nazwy kolumn mają zostać zapisane w pliku (jeśli **TRUE** – będą),
- ***rowNames*** – określa, czy nazwy wierszy mają być dołączone do zapisywanych danych (jeśli **TRUE** – będą),
- ***overwrite*** – określa, czy istniejący plik o tej samej nazwie ma zostać nadpisany (jeśli TRUE – zostanie nadpisany),
- ***firstActiveRow***, ***firstActiveCol*** – parametry pozwalające określić, w którym miejscu arkusza (wiersz i kolumna) powinien rozpocząć się zapis danych.

Przykład 6.3.

Wczytywanie i zapis danych do arkuszy kalkulacyjnych.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **arkusze.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
library(openxlsx)
dane <- data.frame(        #utworzenie prostej ramki danych
  Imie = c("Anna", "Tomasz", "Kasia"),
  Wiek = c(23, 30, 27)
)
      #zapis ramki danych do arkusza kalkulacyjnego – plik dane.xlsx
write.xlsx(dane, file = "dane.xlsx", sheetName = "Ramka")

      #wczytanie danych z arkusza kalkulacyjnego do ramki danych
dane_ark <- read.xlsx("dane.xlsx", sheet = 1)
print(dane_ark)
str(dane_ark)
```

6.4. Wczytywanie danych dołączonych do R

W języku R, a także w wielu pakietach dodatkowych, znajdują się gotowe zestawy danych, które mogą być od razu wykorzystane do analiz, ćwiczeń, czy też testowania kodu. Aby wczytać taki zbiór, stosuje się funkcję: **data(*"nazwa_zbioru"*)**, a w przypadku, gdy dane znajdują się w konkretnym pakiecie – pełna postać to **data(*"nazwa_zbioru", package = "nazwa_pakietu"*)**.

Aby zapoznać się z listą dostępnych zbiorów z podstawowych pakietów, można wykorzystać polecenie **data()** bez argumentów. Wśród często używanych zbiorów można wymienić m.in.: **mtcars** (dane dotyczące osiągów i cech technicznych samochodów), **airquality** (dane o jakości powietrza w Nowym Jorku z lat 70.), **iris** (zestaw pomiarów kwiatów irysa: długość i szerokość działek kielicha oraz płatków dla 150 roślin, pochodzących z trzech gatunków), **Titanic** (dane pasażerów statku Titanic, zawierające m.in. informacje o przeżyciu, płci, wieku i klasie biletu), czy też **USArrests** (statystyki przestępczości w USA z 1973 roku).

Przykład 6.4.

Wczytywanie danych dołączonych do R.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **dane.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
data()     #wyświetlenie listy dostępnych zbiorów danych

      #wczytanie przykładowych zbiorów
data("iris")           #pomiary kwiatów irysa
```

```
data("mtcars")          #dane o samochodach
data("airquality")      #jakość powietrza

     #podgląd struktury zbiorów danych
head(iris)      #pierwsze 6 wierszy
str(iris)
summary(iris)
head(iris)
str(iris)
summary(iris)
head(iris)
str(iris)
summary(iris)
```

6.5. Pobieranie danych z Internetu – wybrane aspekty

Możliwości wykorzystania języka R do pobierania danych z Internetu są dość szerokie. Współcześnie, ogromne ilości informacji są przechowywane online, zarówno w formie strukturalnej, jak i niestrukturalnej. Najprostszą metodą jest importowanie danych z plików tekstowych (np. *.txt, *.csv), czy też arkuszy kalkulacyjnych (np. *.xlsx) hostowanych online, co zostało omówione we wcześniejszych rozdziałach. Aby pobrać pliki z Internetu, można wykorzystać funkcję **download.file()** – wymaga ona podania adresu URL do interesującego zasobu oraz nazwę, pod którą plik zostanie zapisany. W ogólności, tzw. **web scraping** stanowi jednakże dość skomplikowany proces, gdyż obecnie istnieje wiele różnorodnych technologii, które mogą być wykorzystywane w tym zakresie, np. HTML, XML, JSON, czy też interfejsy API.

Dane do analizy w języku R mogą być pozyskiwane bezpośrednio ze stron internetowych utworzonych w języku **HTML** (ang. *HyperText Markup Language*; hipertekstowy język znaczników), który jest wykorzystywany do zdefiniowania struktury i zawartości stron WWW. Do tego celu można wykorzystać pakiety takie jak **XML** lub **rvest**. Dane posiadają strukturę „drzewiastą”, co oznacza, że elementy są zagnieżdżone hierarchicznie i należy je odpowiednio wydobywać. Podobnie zorganizowany jest także tekstowy format **XML** (ang. *Extensible Markup Language*; rozszerzalny język znaczników) służący do opisu i przesyłania danych, który można obsługiwać w R np. dzięki pakietowi **xml2**.

Przykładowo, jeżeli dane są przechowywane w postaci prawidłowo sformatowanej tabeli, można je wydobyć za pomocą funkcji **readHTMLTable()** z pakietu **XML**, która umożliwia wczytanie wszystkich tabel ze wskazanego pliku lub adresu URL. Funkcja

ta może przyjąć argument ***header = TRUE***, który sygnalizuje, że tabela zawiera nagłówek. Wcześniej jednak często wykorzystuje się funkcję **htmlParse()**, która parsuje dokument HTML i generuje strukturę drzewa możliwą do dalszej analizy w R.

Należy jednak pamiętać, że informacje na stronach internetowych bardzo często są rozproszone i nie znajdują się w prostych tabelach. Wówczas przydatny okazuje się pakiet **rvest** autorstwa Hadleya Wickhama, z którym warto się zapoznać: https://rvest.tidyverse.org/. Zawiera on szereg funkcji pozwalających na scrapowanie danych z różnych fragmentów kodu HTML. Funkcja **read_html()** umożliwia wczytanie kodu strony internetowej bezpośrednio z adresu URL, tworząc obiekt typu **xml_document**; zawiera on kod HTML strony WWW. Następnie można użyć funkcji **html_nodes()**, aby wyodrębnić odpowiednie fragmenty dokumentu na podstawie selektorów CSS lub XPath. Funkcja **html_text()** pozwala pobierać tekst znajdujący się w określonych znacznikach, natomiast **html_attr()** daje dostęp do atrybutów znaczników HTML. W przypadku tabel przydatna jest także funkcja **html_table()**, która konwertuje dane do postaci ramki danych w R.

W praktyce pobieranie danych z Internetu nie zawsze jest proste. Wiele stron generuje treść dynamicznie (np. przy wykorzystaniu języka JavaScript lub wielu innych technologii), dlatego klasyczne narzędzia do scrapowania danych mogą być niewystarczające. Istotnym aspektem pracy z danymi z Internetu są również kwestie etyczne i prawne. Nie każdy serwis pozwala na automatyczne wydobywanie danych, więc należy każdorazowo zapoznawać się z istniejącymi uwarunkowaniami. Warto też zadbać o to, by nie przeciążać serwera zbyt dużą liczbą zapytań w krótkim czasie

W przypadku dość uniwersalnego i elastycznego formatu **JSON** (ang. *JavaScript Object Notation*), w R można wykorzystać pakiety takie jak m.in. **jsonlite** (wraz z funkcją wczytującą dane o nazwie **fromJSON()**), czy też **rjson**. Jeżeli jest to możliwe, warto korzystać z istniejących **interfejsów API** udostępnianych przez różnorodne organizacje, dzięki czemu w dość prosty sposób można pozyskiwać interesujące dane. Każde API jest unikalne, więc zawsze należy zapoznać się z jego dokumentacją. Tutaj warto wyróżnić jeszcze jeden, potencjalnie przydatny pakiet: **httr**, który pozwala na bezpośrednią komunikację z interfejsami, umożliwiając wykonywanie zapytań HTTP (GET, POST itp.).

Przykład 6.5.

Pobieranie danych z Internetu w R – wybrane aspekty.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **Internet.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
 library(rvest)
       #pobranie tabeli ze strony WWW (tu: Wikipedia – Miasta w Polsce)
url_html <- "https://pl.wikipedia.org/wiki/Miasta_w_Polsce"
strona <- read_html(url_html)       #wczytanie kodu HTML strony WWW
      #wyodrębnienie wszystkich tabel
tabele <- html_table(html_nodes(strona, "table"))
length(tabele)     #ile tabel znaleziono sumarycznie?
      #pierwsza tabela w postaci tibble (10 największych miast Polski)
head(tabele[[1]])
str(tabele[[1]])

      #pobranie z API NBP kursów walut (euro i dolar)
      #potrzebne pakiety – analogiczne instrukcje warto wykorzystywać
       #w innych przykładach!!!!
if (!require(httr)) {
  install.packages("httr", dependencies = TRUE)
  library(httr)
} else {library(httr)}

if (!require(jsonlite)) {
  install.packages("jsonlite", dependencies = TRUE)
  library(httr)
} else {library(jsonlite)}

      #pobieranie kursów walut USD i EUR z ostatnich 30 dni
url_usd <-
"http://api.nbp.pl/api/exchangerates/rates/A/USD/last/30/?format=json"
url_eur <-
"http://api.nbp.pl/api/exchangerates/rates/A/EUR/last/30/?format=json"
      #zapytania
usd_data <- fromJSON(content(GET(url_usd), "text", encoding = "UTF-8"))
eur_data <- fromJSON(content(GET(url_eur), "text", encoding = "UTF-8"))
      #wyodrębnienie danych z kursami
usd_rates <- usd_data$rates
eur_rates <- eur_data$rates
```

```
      #połączenie w jedną ramkę danych
kursy <- data.frame(
  Data = usd_rates$effectiveDate,
  USD = usd_rates$mid,
  EUR = eur_rates$mid
)
print(kursy)
```

6.6. Wczytywanie danych z innych narzędzi

Dane, które mają zostać wykorzystane w programie R, często nie są gromadzone bezpośrednio w tym środowisku, lecz w różnych innych narzędziach statystycznych (SPSS, SAS, Stata, Octave, Matlab itd.), czy też bazach danych. Aby umożliwić ich dalszą analizę, konieczne jest zaimportowanie ich w odpowiednim formacie. W tym celu w R można wykorzystać jeden z wielu dostępnych pakietów, np. **foreign** lub **haven**.

R umożliwia bezpośrednią komunikację z różnymi systemami baz danych, takimi jak **SQLite**, **PostgreSQL**, czy **MS Access**. Przykładowo, baza **SQLite** jest bardzo lekką i prostą w obsłudze bazą danych, która przechowuje całą swoją zawartość w jednym pliku. Do pracy z nią w R często wykorzystywany jest pakiet **RSQLite**, a samo korzystanie z tej bazy składa się z kilku etapów. Najpierw należy wczytać sterownik do łączenia się z bazą, co można zrobić funkcją **dbDriver()**, a następnie nawiązać połączenie z bazą danych (np. za pomocą funkcji **dbConnect()**). Następnie wysyła się odpowiednie zapytania SQL, w wyniku czego otrzymuje się tabele z danymi. Po zakończeniu pracy istotne jest zwolnienie połączenia z bazą (funkcja **dbDisconnect()**). Do pracy z innymi bazami danych przydatny może być również pakiet **RODBC**.

Przykład 6.6.

Wczytywanie danych z innych narzędzi – **SQLite.**

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **sql.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
install.packages("RSQLite", dependencies = TRUE)
library(RSQLite)
con <- dbConnect(SQLite(), ":memory:")     #utworzenie bazy w pamięci
      #utworzenie tabeli w bazie danych
```

```
dbWriteTable(con, "osoby", data.frame(
  id = 1:3,
  imie = c("Anna", "Tomasz", "Kasia"),
  wiek = c(25, 30, 28)
))
      #pobranie danych za pomocą SQL
wynik <- dbGetQuery(con, "SELECT * FROM osoby WHERE wiek > 26")
print(wynik)
dbDisconnect(con)      #zamknięcie połączenia
```

7. Wybrane techniki przetwarzania i transformacji danych w R

Środowisko R udostępnia ogromną liczbę różnorodnych pakietów i funkcji przeznaczonych do przetwarzania i transformacji danych.

Każdy *inżynier danych* wykorzystujący język R powinien wiedzieć, że do pracy z danymi w tym języku szczególnie przydatna jest kolekcja pakietów tidyverse, którą stanowi zestaw otwartoźródłowych pakietów propagowanych przez Hadleya Wickhama. Jest to spójna kolekcja narzędzi, które dobrze ze sobą współpracują i obejmują cały proces analizy: od wczytania danych, przez ich przekształcanie, aż po wizualizację i modelowanie. W skład **tidyverse** wchodzą m.in. pakiety:

- **readr** – do importu danych z różnych typów plików,
- **tibble** – do przechowywania danych w nowoczesnym, bardziej czytelnym formacie,
- **dplyr** – do wygodnego filtrowania, grupowania i transformacji danych,
- **ggplot2** – do tworzenia wykresów i wizualizacji danych,
- **tidyr** – do porządkowania danych i nadawania im „uporządkowanej” struktury,
- **purrr** – wspierający programowanie funkcyjne,
- **stringr** – ułatwiający pracę z tekstem,
- **lubridate** – przydatny do analizy danych daty i czasu.

Niestety, ramy tego kursu nie pozwalają na dokładniejsze omówienie tego zagadnienia – wspominane są tylko pewne, wybrane funkcjonalności. Nieco bardziej zaawansowane osoby korzystające z języka R powinny koniecznie zapoznać się jednak ze szczegółowymi materiałami na jego temat – https://www.tidyverse.org!

Przy tworzeniu niniejszego rozdziału korzystano z pomocy programu R, dokumentacji R (“Rdocumentation“, 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN (“The Comprehensive R Archive Network“, 2025) i jego podstron (gdzie znajdują się m.in. opisy poszczególnych pakietów i funkcji), dokumentacji Tidyverse (“Tidyverse“, 2025) i jej podstron oraz pozycji literaturowych (Biecek, 2018; Gągolewski, 2014; Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012).

7.1. Iterowyanie po danych za pomocą funkcji z rodziny „apply”

Wśród wielu technik, warto wyróżnić rodzinę funkcji „**apply**”, które umożliwiają iterowanie po zbiorach danych bez konieczności stosowania klasycznych pętli iteracyjnych, takich jak np. **for()**. Różnice pomiędzy poszczególnymi funkcjami wynikają ze sposobu, w jaki definiowane są podzbiory danych, na których wykonywana jest dana operacja – mogą to być:

- elementy macierzy – funkcja **apply()**:
  funkcja **apply()** może być użyta wobec obiektu typu macierzowego, przy czym wszystkie analizowane elementy muszą być więc tego samego typu. Jej ogólna składnia to: **apply(*x*, *MARGIN*, *FUN*, …)**
  gdzie: ***x*** – obiekt (będący macierzą); ***MARGIN*** – margines, względem którego funkcja **FUN** jest stosowana: ***1*** – operowanie na wierszach, ***2*** – operowanie na kolumnach; ***FUN*** – funkcja do zastosowania (np. **sum**, **mean**, **summary**); **…** – opcjonalne argumenty, np. ***na.rm=TRUE*** powoduje usunięcie wartości **NA** przed wykonaniem obliczeń.
  Jeśli funkcja zostanie użyta na ramce danych, wówczas obiekt ten jest automatycznie konwertowany do macierzy w tle,
- elementy listy – funkcja **lapply()**:
  funkcja **lapply()** działa w sposób analogiczny, lecz jej zadaniem jest iteracja po elementach listy. Zwraca ona listę tej samej długości, w której każdy element jest wynikiem zastosowania funkcji ***FUN***. Składnia: **lapply(*x*, *FUN*, ...)**
  Funkcja **lapply()** wymaga podania co najmniej dwóch argumentów: ***x*** (obiektu będącego listą) i ***FUN*** (funkcji do zastosowania),
- elementy wektora – funkcja **sapply()**:
  funkcja **sapply()** działa tak samo jak funkcja **lapply()** oraz posiada analogiczną składnię. Zwraca jednak wynik operacji jako wektor.
- podgrupy wskazane przez jedną lub kilka zmiennych – funkcje **by()** oraz **tapply()**:
  funkcja **tapply()** posiada składnię: **tapply(*x*, *INDEX*, *FUN*)**. Wykonuje ona funkcję ***FUN*** dla każdej podgrupy wektora ***x*** zdefiniowanej przez zmienną czynnikową ***INDEX***.
  Podobne działanie realizuje funkcja **by()**, której ogólna postać wygląda następująco: **by(*x*, *INDEX*, *FUN*)**. W odróżnieniu od funkcji **tapply()**, funkcja **by()** umożliwia przekazanie jako argumentu ***x*** macierzy lub listy, a argument ***INDEX*** może być listą zmiennych grupujących. Wynikiem działania jest obiekt klasy ***by***, ale po usunięciu informacji o klasie, np. poprzez użycie funkcji **unclass()**, otrzymuje się zwykłą macierz.

Podsumowując, wszystkie wymienione powyżej funkcje posiadają argument ***FUN*** – oznaczający operację, która zostaje zastosowana do wybranych fragmentów danych. Należy pamiętać, że w sytuacji, gdy w wektorze pojawia się przynajmniej jedna wartość brakująca **NA**, to wynik standardowych funkcji statystycznych również będzie równy **NA**, o ile nie zostanie jawnie określone usuwanie wartości brakujących (np. przez wykorzystanie argumentu ***na.rm=TRUE***).

Rodzinę funkcji „**apply**" uzupełnia funkcja **mapply()** – wielowymiarowa odmiana funkcji **sapply()**. Pozwala ona stosować funkcję ***FUN*** równocześnie do wielu argumentów wektorów (lub list). Argumentami są: zastosowana funkcja ***FUN*** oraz kilka (dwa lub więcej) wektorów ***W1***, ***W2***, itd. o tej samej długości. Funkcja **mapply()** posiada składnię: **mapply(*FUN*, *W1*, *W2*, …)**.

Przykład 7.1.

Iterowanie po danych za pomocą funkcji z rodziny „apply" – cz. I.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **apply1.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
m <- matrix(1:12, nrow = 3, ncol = 4)    #macierz m
print(m)
apply(m, 1, sum)        #suma wartości w wierszach
rowSums(ma  )           #wynik jw.
apply(m, 2, mean)       #średnia arytmetyczna wartości w kolumnach
apply(m, 1, min)
m[2,2] <- NA            #wstawienie wartości NA na pozycję [2, 2]
print(m)
apply(m, 1, sum)                  #jedna z wartości to NA
rowSums(ma  )                     #wynik jw.
apply(m, 1, sum, na.rm = TRUE)    #jaka jest różnica?

pierwiastek3 <- function(l) {    #funkcja oblicza pierwiastek 3 stopnia
  return(l^(1/3)) }
      #zwróć uwagę na różnice
apply(m, 1, pierwiastek3)
apply(m, 2, pierwiastek3)
sapply(m, pierwiastek3)
lapply(m, pierwiastek3)

lista <- list(a = 1:5, b = 6:10, c = c(2, 4, 6, NA))      #lista
```

```
lapply(lista, mean, na.rm = TRUE)
sapply(lista, mean, na.rm = TRUE)
typeof(lapply(lista, mean, na.rm = TRUE))      #lista
typeof(sapply(lista, mean, na.rm = TRUE))      #wektor

dane <- data.frame(        #utworzenie prostej ramki danych
  Imie = c("Anna", "Tomasz", "Kasia"),
  Wiek = c(23, 30, 27),
  Plec = factor(c("K", "M", "K"))
)
      #średni wiek wg płci
tapply(dane$Wiek, dane$Plec, mean)
by(dane$Wiek, dane$Plec, mean)

mapply(function(a, b) a^2 + b, 1:5, 6:10)     #np. 1^2+6=7
```

Przykład 7.2.

Iterowanie po danych za pomocą funkcji z rodziny „apply" – cz. II. Operacje na ramce danych – zbiór mtcars.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **apply2.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
data("mtcars")    #zbiór danych mtcars
      #średnie wartości dla każdej kolumny (MARGIN = 2)
apply(mtcars, 2, mean)
typeof(apply(mtcars, 2, mean))            #double
is.data.frame(apply(mtcars, 2, mean))     #FALSE
      #suma wartości w każdym wierszu (MARGIN = 1)
apply(mtcars, 1, sum)
      #wartości minimalne i maksymalne w każdej kolumnie
apply(mtcars, 2, function(x) c(min = min(x), max = max(x)))
      #średnie spalanie (mpg) w zależności od liczby cylindrów (cyl)
tapply(mtcars$mpg, mtcars$cyl, mean)
```

7.2. Manipulowanie danymi z pakietem dplyr

Jedno z kluczowych narzędzi wykorzystywanych obecnie w procesach analizy i transformacji danych w języku R to pakiet **dplyr**, który został opracowany przez Hadleya Wickhama. Jego głównym przeznaczeniem jest praca z obiektami typu

data.frame (**ramka danych**), jednak należy podkreślić, że nie ogranicza się wyłącznie do nich – operacje, które można wykonać na ramkach danych, mogą być stosowane także do innych struktur. Pakiet **dplyr** jest integralną częścią kolekcji pakietów **tidyverse**. W dalszej części przedstawiono krótko wybrane funkcjonalności tego pakietu.

7.2.1. Obiekty typu tibble

Jednym z istotniejszych elementów wprowadzonych przez pakiet **dplyr** jest nowy typ obiektu rozszerzający ramki danych – **tibble**. Obiekty tego typu są implementowane jako klasa **tbl_df**, będąca podklasą tradycyjnych ramek danych (**data.frame**). Obiekty typu **tibble** różnią się od klasycznych ramek danych m.in. sposobem wyświetlania (np. prezentują tylko część kolumn i obserwacji w konsoli, co ułatwia czytelność) oraz bardziej przewidywalnym traktowaniem typów danych. Konwersja danych do obiektu typu **tibble** odbywa się za pomocą funkcji **dplyr::as_tibble()**, gdzie operator „podwójny dwukropek" (**::**) pozwala jednoznacznie wskazać pakiet, z którego pochodzi dana funkcja.

7.2.2. Przetwarzanie potokowe

Proces analizy danych niezwykle rzadko kończy się jedynie na wykonaniu pojedynczej operacji – często składa się z wielu kroków, takich jak m.in. filtrowanie, transformacja, grupowanie, czy podsumowania. Kluczowe jest więc, aby kod w języku R pozostawał czytelny i łatwy do zrozumienia. W tym celu pakiet **dplyr** spopularyzował podejście znane jako paradygmat **przetwarzania potokowego**. Idea ta polega na tym, że zamiast zagnieżdżać funkcje w sobie nawzajem, można przekazywać wynik działania jednej operacji bezpośrednio do kolejnej (na wejście, domyślnie jako jej pierwszy argument) za pomocą specjalnego operatora potoku: **%>%**. Zapis **X %>% funkcja(Y)** odpowiada więc instrukcji **funkcja(X, Y)**.
W środowisku **RStudio Desktop** operator ten można wstawić skrótem klawiaturowym: ***Ctrl + Shift + M***. Jeżeli wynik operacji ma być przekazany nie jako pierwszy, lecz jako dalszy argument funkcji, można to zrobić poprzez operator „**.**" , np.: **X %>% funkcja(Y, data = .)**.

7.2.3. Filtrowanie wierszy i kolumn

W pracy z danymi bardzo często zachodzi potrzeba ograniczenia analizowanego zbioru.

Filtrowanie wierszy pozwala wyodrębnić jedynie te obserwacje, które spełniają określone warunki. Kluczowa w tym aspekcie jest funkcja **filter()**, która wybiera (filtruje) wskazane wiersze (obserwacje) ze zbioru danych; wiersze te muszą spełniać określony warunek logiczny lub kilka warunków jednocześnie. Pierwszy argument to zbiór danych, na których funkcja ta ma operować, a kolejne argumenty stanowią warunki logiczne. W wyniku działania funkcji zwracane są te wiersze, które spełniają wszystkie podane warunki logiczne. Tworząc warunki logiczne można wykorzystywać nazwy kolumn ze zbioru danych bez podawania dodatkowych odnośników.

Funkcja **distinct()** usuwa zduplikowane wiersze. Z kolei, funkcje **slice_min(*zmienna*, *n* = *x*)** i **slice_max(*zmienna*, *n* = *x*)** wybierają ***x*** wierszy z najniższymi lub najwyższymi wartościami zmiennej ***zmienna***. Funkcje **sample_frac()** i **sample_n()** służą z kolei do losowego pobierania próbek wierszy z ramki danych – w **sample_frac()** podaje się określoną wartość procentową (argument ***size*** – liczba w przedziale 0-1, np. 0.2 = 20% obserwacji), a w **sample_n()** dokładną liczbę losowanych wierszy (argument ***size***). W oby przypadkach przydatny jest także argument ***replace***, stanowiący o tym, czy losować z powtórzeniami (**TRUE**), czy też bez powtórzeń (**FALSE**).

Filtrowanie kolumn polega z kolei na ograniczeniu liczby analizowanych zmiennych, co oznacza wybór jedynie tych kolumn, które są istotne w danym momencie analizy i pominięcie pozostałych. W tym zakresie najistotniejsza jest funkcja **select()**, która wybiera wskazane kolumny (zmienne) ze zbioru danych. Pierwszy argument to zbiór danych, na których funkcja ta ma operować, a kolejnymi są nazwy kolumn, które mają zostać wybrane. Dopuszczalne jest również wykorzystanie operatora negacji „**-**” – wtedy wybierane są wszystkie kolumny poza wskazanymi. Pomocne są również funkcje wybierające kolumny o nazwach spełniających określone warunki:

- **contains()** – nazwa obejmuje podany ciąg znaków,
- **starts_with()** – nazwa zaczyna się od podanego ciągu znaków,
- **ends_with()** – nazwa kończy się podanym ciągiem znaków,
- **matches()** – dopasowuje wskazane wyrażenie regularne.

7.2.4. Transformacje zmiennych, sortowanie i statystyki w podgrupach

W trakcie analizy danych i tworzenia modeli, często zachodzi potrzeba utworzenia nowych zmiennych w oparciu o te już istniejące. Do tego szczególnie przydatne mogą być funkcje:

- **mutate()** – modyfikuje zmienne; dodaje nową kolumnę lub zmienia już istniejące; pozwala dodać nowe kolumny do istniejącego zbioru danych, zachowując wszystkie dotychczasowe kolumny,
- **transmute()** – działa podobnie do funkcji **mutate()**, ale w efekcie końcowym pozostawia tylko nowo utworzone kolumny; wszystkie istniejące kolumny są usuwane,
- **across() –** umożliwia zastosowanie tej samej transformacji do wielu kolumn jednocześnie, pozwalając na zastosowanie pewnej semantyki wewnątrz funkcji takich jak **summarise()** i **mutate()**.

Sortowanie danych według wybranej kolumny jest także szeroko wykorzystywane podczas procesu eksploracji danych. Do tego celu służy funkcja **arrange()**, która pozwala na sortowanie po jednej lub kilku kolumnach. Jeśli pojawią się „remisy" względem pierwszej kolumny, porządek ustalają kolejne kolumny. Aby odwrócić kolejność sortowania, można użyć funkcji **desc()**.

Często istnieje również konieczność wyznaczania statystyk lub agregatów w pewnych podgrupach. W pakiecie **dplyr** kluczowe są dwie funkcje:

- **group_by()** – definiuje grupy, w których będą przeprowadzane obliczenia (wyznaczane statystyki). Nie powoduje ona żadnych modyfikacji danych, tylko dodaje sam znacznik grupowania (informacja o zmiennej grupującej), który ma istotny wpływ na wiele funkcji, np.
  - w przypadku funkcji **summarise()** powoduje wyliczenie statystyk dla każdej możliwej kombinacji zmiennych grupujących,
  - w przypadku funkcji **arrange()**, powoduje, że wiersze są sortowane najpierw po zmiennej grupującej,
  - w przypadku funkcji **sample_n()** powoduje, że próbkowanie jest wykonywane niezależnie w każdej podgrupie,
- **summarise()** – pozwala obliczać różne agregaty, np. średnią arytmetyczną, sumę, czy też liczbę wierszy (funkcja **n()**).

Przykład 7.3.

Manipulowanie danymi z pakietem dplyr – wybrane aspekty.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **dplyr.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
data("mtcars")      #zbiór danych mtcars
head(mtcars)
summary(mtcars)
str(mtcars)
typeof(mtcars)
print(mtcars)
mtcars_tibble <- as_tibble(mtcars)     #konwersja na typ tibble
summary(mtcars_tibble)
str(mtcars_tibble)
typeof(mtcars_tibble)
print(mtcars_tibble)   #zwróć uwagę na różnice w stosunku do print(mtcars)

      #wybór kolumn
auta <- select(mtcars, mpg, cyl, hp, gear)   #wybrane 4 kolumny
head(auta)
mtcars %>% select(-mpg) %>% head(n=4)        #wszystkie kolumny oprócz mpg
mtcars %>% select(starts_with("c")) %>% head(n=4)  #nazwa zaczyna się od c

      #filtrowanie – tylko samochody "mocne" (hp > 100)
mocne <- filter(mtcars, hp > 100)
head(mocne)

      #ile jest samochodów o 6 cylindrach i mocy > 100?
mtcars %>% filter(cyl == 6, hp > 100) %>% nrow

mtcars %>% nrow                          #ile obserwacji? – 32
mtcars %>% sample_frac(0.5) %>% head     #losowanie 50% (0.5) obserwacji
mtcars %>% sample_frac(0.5) %>% nrow     #0.5*32=16
mtcars %>% sample_n(5) %>% head          #wylosowanie 5 obserwacji
mtcars %>% sample_n(5) %>% nrow          #5 wierszy
mtcars %>% slice(2:4) %>% head           #wybór wierszy od 2 do 4
mtcars %>% slice(2:4) %>% nrow           #3 wiersze
mtcars %>% slice_max(hp, n=2)            #wybór 2 wierszy z najwyższymi
                                         #wartościami zmiennej hp
```

```
mtcars %>% slice_min(hp, n=2)            #wybór 2 wierszy z najniższymi
                                         #wartościami zmiennej hp

      #sortowanie wg spalania (mpg) - malejąco
posortowane <- arrange(mtcars, desc(mpg))
head(posortowane)

      #nowa kolumna – stosunek mocy do masy
auta2 <- mutate(mtcars, moc_na_mase = hp / wt)
head(auta2)       #istniejące kolumny + "nowa"
auta3 <- transmute(mtcars, moc_na_mase = hp / wt)
head(auta3)       #tylko "nowa" kolumna

      #średnie spalanie wg liczby cylindrów
spalanie <- mtcars %>%
  group_by(cyl) %>%
  summarise(srednie_mpg = mean(mpg))
print(spalanie)

      #przetwarzanie potokowe (%>%) – przykład praktyczny
wynik <- mtcars %>%
  filter(gear == 4) %>%
  mutate(moc_na_mase = hp / wt) %>%
  arrange(desc(moc_na_mase)) %>%
  select(mpg, cyl, hp, wt, moc_na_mase)
head(wynik)

      #średnia moc w zależności od liczby cylindrów i liczba pojazdów
      #w powstałych podgrupach
mtcars %>% group_by(cyl) %>% summarise(Średnia_moc = mean(hp), Ile=n())
```

7.3. Inne techniki transformacji danych. Pakiet tidyr

Poza rodziną funkcji **apply** oraz pakietem **dplyr**, w języku R istnieje szereg innych mechanizmów umożliwiających transformację danych. Nie sposób przedstawić je w niniejszym opracowaniu.

Jednym z bazowych rozwiązań jest funkcja **merge()**, działająca podobnie do SQL-owego polecenia **join**. Pozwala ona połączyć dwie ramki danych lub macierze, wskazując kolumny (klucze), na podstawie których identyczne obserwacje są dopasowywane i scalane w jeden obiekt.

Bardzo ważnym elementem są również funkcje dostępne w pakiecie **tidyr**, zaprojektowanego do przekształcania (kształtowania/ transformacji) formatu danych. Jest on niezwykle przydatny w sytuacji, gdy dane nie są zapisane w formie „tabelarycznej" (tj. wiersze jako obserwacje, kolumny jako zmienne, wartości znajdujące się w pojedynczych komórkach). Przykłady przydatnych funkcji z pakietu **tidyr**:

- **unite()** – umożliwia scalanie wartości z kilku kolumn w jedną,
- **separate()** – odwrotna operacja do **unite()**; rozdziela jedną kolumnę na kilka, według zdefiniowanego separatora,
- **spread()** – przekształca dane z postaci „wąskiej" do „szerokiej" (więcej kolumn). Pierwszy argument to zbiór danych, drugi – kolumna klucza, trzeci – wartości, które zostaną wpisane do kolumn,
- **gather()** – odwrotna operacja do **spread()**; zbiera wiele kolumn w jedną,
- **drop_na()** – usuwa wiersze zawierające braki danych,
- **replace_na()** – zastępuje brakujące wartości (**NA**) wskazanymi danymi.

Przykład 7.4.

Wybrane techniki transformacji danych – funkcja merge().

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **merge.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
library(dplyr)
      #pierwsza ramka danych – podstawowe dane o pracownikach
pracownicy <- data.frame(
  ID = c(10, 11, 12),
  Nazwisko = c("Nowak", "Kowalska", "Wiśniewski")
)
      #druga ramka danych – dodatkowa informacja: rok zatrudnienia
zatrudnienie <- data.frame(
  ID = c(10, 12),
  Rok_zatrudnienia = c(2015, 2018)
)
      #trzecia ramka danych – miejscowość zamieszkania pracowników
miasto <- data.frame(
  ID = c(10, 11, 12),
  Miasto = c("Poznań", "Gdańsk", "Łódź")
)

      #połączenie wszystkich ramek danych według kolumny ID
```

```
dane_polaczone <- merge(pracownicy, zatrudnienie, by = "ID", all.x = TRUE)
%>% merge(miasto, by = "ID", all.x = TRUE)
print(dane_polaczone)
```

Przykład 7.5.

Wybrane techniki transformacji danych – pakiet tidyr.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **tidyr1.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
library(tidyr)

     #prosta ramka danych z imieniem i nazwiskiem w osobnych kolumnach
studenci <- data.frame(
  imie = c("Anna", "Bartek", "Celina"),
  nazwisko = c("Nowak", "Kowalski", "Wiśniewska"),
  ocena = c(5, NA, NA)
)
print(studenci)

     #scalenie kolumn imie i nazwisko w jedną kolumnę nazwa
osoby_full <- unite(studenci, col = "nazwa", imie, nazwisko, sep = " ")
print(osoby_full)

     #rozdzielanie kolumny z imieniem i nazwiskiem
osoby_sep <- separate(osoby_full, nazwa, into = c("imie", "nazwisko"), sep
= " ")
print(osoby_sep)

     #usuwanie wierszy z brakami danych (NA)
bez_brakow <- drop_na(studenci)
print(bez_brakow)

     #zastępowanie braków w danych (NA) wartością 2
bez_brakow <- replace_na(studenci, list(ocena = 2))
print(bez_brakow)
```

Przykład 7.6.

Wybrane techniki transformacji danych – pakiet tidyr i zbiór danych mtcars.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **tidyr2.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
data("mtcars")
      #dodanie kolumny model z nazwami samochodów
mt <- data.frame(model = rownames(mtcars), mtcars)

      #wybór kilku kolumn liczbowych do transformacji
dane <- mt[, c("model", "mpg", "hp", "wt")]
print(head(dane))

      #funcja gather(): wynik w postaci (zmienna, wartość)
dane_g <- gather(dane, key = "zmienna", value = "wartosc", -model)
print(dane_g)

      #funkcja spread() – odwrotna operacja do gather()
dane_s <- spread(dane_long, key = "zmienna", value = "wartosc")
print(dane_s)
```


Politechnika Świętokrzyska
Kielce University of Technology
Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki”
nr FERS.01.05-IP.08-0234/23

8. Praca z danymi tekstowymi w R – wybrane aspekty

Wektory napisów, wybrane znaki specjalne oraz funkcja **format()** zostały już omówione w podrozdziale 3.5. Warto przypomnieć, że obiekty typu **character** w R przechowują napisy (łańcuchy znaków). Wprowadzanie napisów rozpoczyna się od pojedynczego apostrofu (**'**) lub podwójnego cudzysłowu (**"**) i musi być zakończone tym samym znakiem. W łańcuchach mogą znajdować się dowolne znaki, w tym znaki specjalne. Do ich wyświetlania można wykorzystać funkcje **print()** oraz **cat()**. Należy pamiętać, że w języku R napisy traktowane są jako wektory całych łańcuchów znaków, a więc w praktyce ma się do czynienia z „wektorem napisów".

Przy tworzeniu niniejszego rozdziału korzystano z pomocy programu R, dokumentacji R ("Rdocumentation", 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN ("The Comprehensive R Archive Network", 2025) i jego podstron (gdzie znajdują się m.in. opisy poszczególnych pakietów i funkcji), dokumentacji Tidyverse ("Tidyverse", 2025) i jej podstron oraz pozycji literaturowych (Biecek, 2018; Gągolewski, 2014; Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012), a także ("Character encoding", 2025; "Regular expression", 2025).

8.1. Standardy kodowania znaków

Na poziomie sprzętowym komputery nie operują bezpośrednio na znakach, lecz na liczbach. Zasadne jest więc pytanie: *Jak przechowywane w pamięci są znaki, które można wydrukować w konsoli R?* Do reprezentacji znaków używany jest m.in. standard **ASCII** (ang. *American Standard Code for Information*), w którym każdej liczbie od 0 do 127 przyporządkowane są ściśle zdefiniowane znaki (2^7=128 znaków); kody od 1 do 31 to tzw. znaki kontrolne, natomiast pozostałe reprezentują litery, cyfry i znaki specjalne – rys. 13. W systemie **ASCII** istnieje ustalony porządek leksykograficzny: $\mathbf{0} < \mathbf{1} < \mathbf{9} < \boldsymbol{A} < \boldsymbol{B} < \boldsymbol{Z} < \boldsymbol{a} < \boldsymbol{b} < \boldsymbol{z}$. W konsekwencji możliwe jest porównywanie napisów na podstawie wartości ich kodów liczbowych. Kod ***i***-tej wielkiej litery to „A"+***i***-1. Kod ***i***-tej małej litery to „a"+***i***-1. Dla przykładu: zamiana dużej litery ***K*** na małą literę jest możliwa za pomocą operacji „***K***"+32_{10}, a zamiana małej litery ***d*** na dużą literę jest możliwa za pomocą operacji „***d***"-32_{10}.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Rys. 13. Tabela wartości kodów ASCII (*Wikimedia Commons,* https://commons.wikimedia.org/wiki/File:ASCII-Table-wide.svg#/media/File:ASCII-Table-wide.svg, Public domain)

[Tekst alternatywny. Rysunek 13 przedstawia tabelę kodów ASCII – kompletny zestaw znaków wraz z ich kodami numerycznymi. Obraz został udostępniony w domenie publicznej.]

Problemy z kodowaniem **ASCII** pojawiają się w momencie, gdy zajdzie potrzeba posłużenia się znakami spoza jego podstawowego zestawu, np. polskimi literami takimi jak ą, ć, ś czy ź. W takiej sytuacji można wykorzystać jeden z 8-bitowych standardów (gdzie każdy znak reprezentuje jeden bajt, tj. liczbę całkowitą z zakresu od 0 do 255), obejmujących dodatkowo znaki z alfabetów wschodnioeuropejskich. Do najczęściej wykorzystywanych należą **ISO-8859-2** oraz **Windows-1250**, przy czym ten drugi był przez długi czas domyślnym kodowaniem w systemach Windows. W obu tych standardach zakres kodów od 0 do 127 jest zgodny z klasycznym **ASCII**.

Ponieważ użytkownicy komputerów i Internetu posługują się wieloma językami, w naturalny sposób powstała potrzeba stworzenia systemu bardziej uniwersalnego. Tak narodziła się rodzina kodowań **Unicode**, umożliwiająca obsługę większości

znaków używanych na świecie. Najpopularniejszym przedstawicielem jest **UTF-8** (ang. *8-bit Unicode Transformation Format*), w którym pojedynczy znak może być zapisany z użyciem od 1 do 4 bajtów. Warto zauważyć, że w zakresie od 0 do 127 **UTF-8** zachowuje pełną zgodność z **ASCII**, dzięki czemu jest z nim wstecznie kompatybilny. Dzięki takiej konstrukcji **UTF-8** pozwala na reprezentację znacznie większej liczby symboli niż tradycyjne kodowania 8-bitowe. **UTF-8** jest obecnie najczęściej stosowanym formatem w wielu systemach operacyjnych oraz w komunikacji sieciowej. Do przekształcenia napisu zakodowanego w **UTF-8** na wektor typu **integer**, w języku R, służy funkcja **utf8ToInt()**; **intToUtf8()** wykonuje operację odwrotną

Każdy użytkownik R powinien pamiętać, że dostępne kodowania zależą od konfiguracji systemu operacyjnego. Ich listę można uzyskać za pomocą polecenia **iconvlist()**, a konwersję między różnymi systemami znaków umożliwia funkcja **iconv()**. Warto podkreślić, że niezależnie od przyjętego standardu, napis w pamięci komputera jest reprezentowany jako wektor liczb całkowitych, co pozwala na jego dalsze przetwarzanie – algorytmy przetwarzania napisów sprowadzają się do wykonywania pewnych działań na odpowiednich wektorach liczbowych. Można ująć to w ten sposób, że pojedynczy napis w pamięci komputera jest reprezentowany jako wektor liczb całkowitych odpowiadających kodom znaków; wektor napisów w R, czyli obiekt typu **character**, stanowi zbiór takich wektorów.

Przykład 8.1.

Praca z danymi tekstowymi w R – podstawowe aspekty

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **napisy1.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
napis1 <- "ID"                      #tworzenie napisów
napis2 <- 'Język R'
print(napis1)
cat(napis1,"\n", napis2)
head(iconvlist(), n = 20)           #lista dostępnych kodowań
utf8ToInt("Inżynieria danych")      #kody znaków UTF-8
intToUtf8(c(73,68))                 #znaki o podanych kodach
```

8.2. Podstawowe operacje na tekście

Dane tekstowe, obok danych liczbowych, stanowią bardzo często wykorzystywany typ informacji – w praktyce analizy danych niezbędne jest więc opanowanie podstawowych narzędzi służących do manipulowania łańcuchami znaków.

8.2.1. Funkcje wbudowane

W R dostępnych jest wiele funkcji przeznaczonych do pracy z tekstem. Wśród tych podstawowych można wyróżnić:

- **paste()** – skleja (łączy) napisy; argument ***sep*= " "** służy do określania separatora, którym łączone są wektory wejściowe. Jeśli określi się argument ***collapse*= " "** , wtedy elementy tego wektora zostaną połączone,
- **paste0()** – odmiana funkcji **paste()**, gdzie domyślnym separatorem jest **""**,
- **nchar()** – zwraca liczbę znaków w napisie,
- **toupper()** – zamienia litery małe na wielkie,
- **tolower() –** zamienia litery wielkie na małe,
- **chartr(*old*, *new*, *wektor*)** – transliteracja znaków; pozwala na zamianę znaków wymienionych w argumencie ***old*** na te, które są zamieszczone w argumencie ***new*** (w każdym elemencie ***wektora***),
- **strsplit(*x*, *wzorzec*)** – dzieli łańcuch znaków ***x*** na podłańcuchy, które są rozdzielone określonym wzorcem,
- **substr(*x*, *start*, *stop*)** – w wyniku działania powstaje podłańcuch łańcucha znaków ***x***, od znaku ***start***, do znaku ***stop***,
- **agrep()**, **sub()**, **gsub()**, **grep()**, **grepl()**, **regexpr()**, **gregexpr()**, **regexec()** – wybrane funkcje operujące na tekście z wykorzystaniem wzorca/ wyrażeń regularnych.

Na wektorach napisów można stosować operatory relacyjne, dzięki czemu można na nich używać m.in. funkcje **sort()** i **unique()**,

Przykład 8.2.

Podstawowe operacje na tekście.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **napisy2.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
napisy <- c("R jest super", "Analiza danych", "Kodowanie UTF-8")
print(napisy)
```

```
paste(napisy, collapse = " | ")    #sklejenie z separatorem
paste("Dzień", "dobry", sep = "_")
paste0("R", "studio")              #sklejenie bez separatora
nchar(napisy)                      #liczba znaków
toupper(napisy)                    #zamiana na duże litery
tolower(napisy)                    #zamiana na małe litery
chartr("aeiou", "12345", "Ala ma kota")  # transliteracja
strsplit(napisy, " ")              #podział wg separatora
substr(napisy, 1, 3)               #podłańcuch znaków

wektor <- c("abc123", "ABC", "aBc", "123")
grep("^[a-z]+", wektor, value = TRUE)   #małe litery na początku
grepl("\\d", wektor)                    #czy zawiera cyfrę?
sub("abc", "XYZ", "abc123")             #zamiana pierwszego dopasowania
gsub("[0-9]", "#", "a1b2c3")            #zamiana wszystkich cyfr
```

8.2.2. Pakiet stringr

Do operacji na tekstach w R szczególnie przydatny może być pakiet **stringr** autorstwa Hadleya Wickhama. Jego funkcje mają ujednoliconą składnię, a ich nazwy rozpoczynają się od przedrostka **str**_. Wśród wielu funkcji warto wymienić:

- **str_sub(*x*, *start* = …, *end* = …)** – służy do wydzielania fragmentów tekstu,
- **str_c(), strj_join()** – pozwalają na łączenie napisów (podobnie jak **paste(),** ale w nieco odmienny sposób),
- **str_length()** – zwraca liczbę znaków w napisie (podobnie jak **nchar()**),
- **str_sort()** – umożliwia sortowanie napisów,
- **str_to_upper()**, **str_to_lower()** – pozwalają na zmianę wielkości liter w napisie,
- **str_trim()** – usuwa białe znaki z początku i końca łańcucha.

Bardzo ważne zagadnienie stanowią wyrażenia regularne (wprowadzone zapewne na innych przedmiotach w trakcie nauki na kierunku *inżynieria danych*). **Wyrażenia regularne** (ang. *regular expressions* lub *regex*) stanowią niezwykle potężne narzędzie do opisywania i wyszukiwania wzorców w tekście; w R domyślnie stosowany jest mechanizm ERE (ang. *extended regular expressions*, standard ***POSIX***). Dzięki nim możliwe jest nie tylko znajdowanie tekstów spełniających określone kryteria, ale również ich zastępowanie. Obrazowo mówiąc, problem wyszukiwania wzorca można sprowadzić do tego, aby orzec, czy w danym napisie występują określone prawidłowości (ew. ze wskazaniem pozycji), np. ustalony napis, napis składający się z dużych liter alfabetu łacińskiego, napis określający numer

telefonu, czy też napis określający adres e-mail. Wyrażenia regularne są powszechnie stosowane w wielu dziedzinach informatyki. Należy jednak zwrócić uwagę na fakt, że pewne znaki mają znaczenie specjalne, dlatego w razie potrzeby można wykorzystać funkcję **fixed()**, która wymusza dopasowanie „znak po znaku", ignorując metaznaki. W ogólności, wyrażenia regularne opierają się na potrzebie stosowania szeregu operatorów – tzw. **metaznaków** (np. **.** – dowolny znak prócz nowej linii; **^** – początek tekstu (linii); **$** – koniec tekstu (linii), **[]** – zawiera dozwolone znaki; **[^]** – zawiera niedozwolone znaki). Gdy zachodzi konieczność użycia tych znaków dosłownie (np. **^**), należy je poprzedzić znakiem ****.

Wybrane funkcje z pakietu **stringr** wykorzystujące wyrażenia regularne to:

- **str_detect()** – sprawdza, czy napis zawiera określony wzorzec; argument ***pattern*** definiuje pożądany wzorzec tekstowy; rezultatem działania jest wektor logiczny określający, które elementy badanego wektora zawierają zdefiniowany wzorzec. Uwaga: należy pamiętać, że we wzorcu wielkość liter jest rozróżniana,
- **str_count()** – zlicza, ile razy dany wzorzec występuje w napisie,
- **str_extract()** – zwraca pierwszy fragment dopasowany do wzorca,
- **str_extract_all()** – zwraca wszystkie fragmenty dopasowane do wzorca,
- **str_locate()** – zwraca pozycję, na której występuje pierwszy raz wzorzec,
- **str_locate_all()** – zwraca wszystkie pozycje, na których występują dopasowane wzorce,
- **str_replace()** – zastępuje pierwszy fragment pasujący do wzorca innym ciągiem znaków,
- **str_replace_all()** – zastępują wszystkie fragmenty pasujące do wzorca innym ciągiem znaków,
- **str_split()** – dzieli napis na fragmenty według podanego wzorca.

Przykład 8.3.

Podstawowe operacje na tekście – pakiet stringr.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **stringr.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
library(stringr)
txt <- c("Ala ma kota", " Kot ", "123-456-789")
str_sub(txt, 1, 3)           #fragmenty tekstu
str_c("Ala", "ma", "kota", sep = "_")
str_length(txt)              #liczba znaków
str_sort(txt)               #sortowanie
```

```
str_to_upper(txt)           #zamiana na wielkie litery
str_to_lower(txt)           #zamiana na małe litery
str_trim(txt)               #usuwa spacje z początku i końca
str_detect(txt, "kot")      #sprawdza, czy zawiera "kot"?
str_count(txt, "[0-9]")     #liczba cyfr
str_extract(txt, "[0-9]+")  #pierwsza liczba
str_extract_all(txt, "[0-9]+")       #wszystkie liczby
str_locate(txt, "ma")               #pozycja pierwszego dopasowania "ma"
str_locate_all(txt, "[0-9]+")     #pozycje wszystkich fragmentów z cyframi
str_replace(txt, "Ala", "Ola")       #zamiana pierwszego wystąpienia "Ala"
str_replace_all(txt, "[0-9]", "X")   #zamiana wszystkich cyfr na "X"
str_split("imię;nazwisko;wiek", ";") #podział tekstu wg separatora ";"
```

9. Podstawy grafiki w R

Wykresy (różnego typu – np. kołowy, punktowy, liniowy, słupkowy) służą do graficznej prezentacji danych. Środowisko R oferuje ogromny zestaw narzędzi do tworzenia grafiki, począwszy od prostych funkcji bazowych, aż po bardziej zaawansowane mechanizmy niskopoziomowe dostępne w pakiecie **graphics**. Ogromną zaletą jest to, że nawet przy podstawowych ustawieniach generowane wykresy charakteryzują się wysoką jakością i przejrzystością. Do wizualizacji danych można wykorzystywać zarówno wbudowane funkcje R, które są intuicyjne w użyciu i dają szybkie rezultaty, jak również rozbudowane pakiety, np. **ggplot2,** pozwalające na tworzenie bardziej złożonych wykresów.

Przy tworzeniu niniejszego rozdziału korzystano z pomocy programu R, dokumentacji R (“Rdocumentation“, 2025) i jej podstron, dokumentacji (*manuals*) znajdującej się w oficjalnym repozytorium CRAN (“The Comprehensive R Archive Network“, 2025) i jego podstron (gdzie znajdują się m.in. opisy poszczególnych pakietów i funkcji), dokumentacji Tidyverse (“Tidyverse“, 2025) i jej podstron oraz pozycji literaturowych (Biecek, 2016; Biecek, 2018; Gągolewski, 2014; Lander, 2018; Medeiros, 2018; Mount, 2022, Nowosad, 2020; Nwanganga, Chapple, 2022; Walesiak, Gatnar, 2012).

9.1. Funkcja plot. Personalizacja wykresów

Jednym z podstawowych narzędzi do tworzenia wykresów w R jest funkcja **plot()** o składni: **plot*(dane_X*, *dane_Y*, …)**
gdzie: ***dane_X*** – współrzędne punktów na osi **X**; ***dane_Y*** – współrzędne punktów na osi **Y**. Długości obu przekazywanych zbiorów (wektorów) danych muszą być takie same. Jeżeli funkcja zostanie wywołana tylko z jednym argumentem, to dane traktowane są jako wartości na osi **Y**, natomiast oś **X** zostaje automatycznie wypełniona kolejnymi liczbami naturalnymi.

Funkcja **plot()** pozwala generować różne typy wykresów, za co odpowiada argument ***type***, np.:

- "***l***" – wykres liniowy,
- "***p***" – wykres punktowy,
- "***b***" – połączenie punktów i linii,
- "***s***" – wykres schodkowy,
- "***h***" – pionowe kreski.

Dostępne są także inne opcjonalne argumenty, które pozwalają spersonalizować wygląd wykresu, np.:

- ***main*** – tytuł wykresu,
- ***xlab***, ***ylab*** – opisy osi,
- ***pch*** – typ symboli punktów (wartość liczbowa od 0 do 25); np. ***0*** – kwadrat; ***1*** – okrąg, ***2*** – trójkąt, ***3*** – znak plus,
- ***lty*** – styl linii; np. ***0*** – brak linii; ***1*** – linia ciągła; ***2*** – linia przerywana,
- ***lwd*** – grubość linii,
- ***col*** – kolor elementów; kolory w R można określać za pomocą nazwy, wektora kilku barw (np. ***col = c("orange","blue","red")***), czy też za pomocą kodu RGB – np. ***col = "#BB01FA"***. Ponadto można korzystać z funkcji generujących palety, takich jak np. **rainbow()**, czy też dedykowanych pakietów, np. **RColorBrewer**. Nazwy kolorów dostępnych w R można sprawdzić za pomocą funkcji **colors()**,
- ***cex*** – skala wielkości punktów.

Warto zauważyć, że funkcja **plot()** może także służyć do wizualizacji funkcji matematycznych – tj. do tworzenia wykresów.

Uwaga: wywołanie funkcji **plot()** usuwa wcześniej utworzony wykres, ale do już istniejącego wykresu można dodawać kolejne elementy przy wykorzystaniu funkcji takich jak:

- **lines()** – dorysowanie linii,
- **points()** – dodanie nowych punktów,
- **abline()** – wstawienie linii,
- **text()** – dodanie tekstu w dowolnym miejscu wykresu.

Dodanie legendy do wykresu jest możliwe poprzez wywołanie osobnej funkcji – **legend()**, która posiada wiele różnorodnych argumentów, np.:

- ***x*** (lub ***x***, ***y***) – miejsce wyświetlenia legendy; można podać współrzędne albo użyć słów kluczowych, takich jak np. : "***bottomright***", "***bottom***", "***bottomleft***", "***center***", "***left***", "***topleft***", "***top***", "***topright***", "***right***",
- ***title*** – tytuł legendy,
- ***box.lty***, ***box.lwd***, ***box.col*** – odpowiednio: typ, grubość i kolor ramki otaczającej legendę,
- ***legend*** – tekst opisujący poszczególne elementy legendy,
- ***cex*** – wielkość czcionki,
- ***lty*** – style linii wyświetlanych w legendzie; np. ***0*** – brak linii; ***1*** – linia ciągła; ***2*** – linia przerywana,

- ***pch*** – style symboli/punktów ; np. ***0*** – kwadrat; ***1*** – okrąg, ***2*** – trójkąt, ***3*** – znak plus,
- ***bg*** – kolor tła legendy,
- ***col*** – kolory elementów,
- ***fill*** – kolory wypełnienia pól składowych legendy,
- ***text.font*** – styl czcionki tekstu legendy; np. ***1*** – normalny, ***2*** – pogrubiony, ***3*** – kursywa, ***4*** – pogrubiona kursywa.

Przykład 9.1.

Funkcja plot. Personalizacja wykresów.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **wykresy1.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
x <- 1:5
y <- c(2, 3, 5, 7, 11)
plot(x, y)                  #wykres punktowy
plot(x, y, type = "l")      #wykres liniowy
plot(x, y, type = "p")      #wykres punktowy
plot(x, y, type = "b")      #wykres punktowo-liniowy
plot(x, y, type = "s")      #wykres schodkowy
plot(x, y,, type = "S")     #wykres schodkowy odwrócony
plot(x, y, type = "h")      #wykres kreskowy (linii pionowych)

colors() %>% head(n=50)     #przykładowe, dostępne kolory

plot(x, y, type = "l",      #wykres liniowy z tytułem i opisem osi
     main = "Wykres liniowy",
     xlab = "Oś X",
     ylab = "Oś Y")

      #personalizacja kolorów i punktorów + legenda
plot(x, y, type = "b", pch = 19, col = "blue",
     main = "Punkty + linie", xlab = "Indeks", ylab = "Wartości")
      #dodatkowe elementy na już istniejącym wykresie
lines(x, y^1.2, col = "red")      #dodatkowa linia
points(x, y^0.8, col = "green")   #dodatkowe punkty
legend("topleft", legend = c("y", "y^1.2", "y^0.8"),   #legenda
       col = c("black", "red", "green"), lty = 1, pch = c(1, NA, NA))
```

```
y1 <- x * 2
y2 <- x * 3
      #pierwszy wykres (linia ciągła niebieska)
plot(x, y1, type = "l", col = "blue", lwd = 2,
     main = "Przykładowy wykres w R",
     xlab = "Oś X", ylab = "Oś Y")
    #dodajemy drugą linię (czerwona, przerywana)
lines(x, y2, col = "red", lty = 2, lwd = 2)
    #dodanie legendy
legend("topleft",                          #położenie legendy
       legend = c("y = 2x", "y = 3x"),     #opisy składowych
       col = c("blue", "red"),             #kolory
       lty = c(1, 2),                      #style linii
       lwd = 2,                            #grubość linii
       title = "Funkcje")                  #tytuł legendy
```

9.2. Wykresy różnych typów

Oprócz funkcji **plot()**, w R dostępnych jest wiele innych funkcji pozwalających na tworzenie różnorodnych wykresów, np.:

- **wykres słupkowy** – funkcja **barplot()** rysuje pionowe lub poziome słupki; wybrane argumenty:
  - pierwszy to wektor wartości do przedstawienia (lub dwuwymiarowa macierz),
  - ***main*** – tytuł wykresu,
  - ***xlab***, ***ylab*** – tytuły osi (odpowiednio X i Y),
  - ***horiz*** – definiuje, czy słupki mają być rysowane poziomo (***TRUE***), czy też pionowo (***FALSE***),
  - ***las*** – określa kierunek, w jakim którym są rysowane opisy przy osiach,
  - ***col*** – kolor słupków,
  - ***add = TRUE*** – nie tworzy nowego wykresu – dorysowuje do już istniejącego.

  Legendę można dodać za pomocą funkcji **legend()**.
- **histogram** – funkcja **hist()** ilustruje rozkład zmiennej ilościowej (przy wykorzystaniu długości słupków, prezentowana jest liczba obserwacji w zadanych przedziałach); wybrane argumenty:
  - pierwszy to wektor liczb,
  - ***freq*** i ***probability*** – określają, czy w trakcie tworzenia histogramu mają być zaznaczane frakcje, czy też liczebności obserwacji,
  - ***breaks*** – określa, w jaki sposób dane mają być podzielone na przedziały w histogramie; może przyjąć wartość liczbową wskazującą

konkretną liczbę przedziałów albo nazwę jednego z dostępnych algorytmów ("***Sturges***", "***Scott***", "***FD***", "***Freedman-Diaconis***"). Jeśli nie zostanie ustawiony, R sam dobierze liczbę przedziałów,

- **wykres pudełkowy** – funkcja **boxplot()** umożliwia wizualizację mediany, kwartylów oraz wartości odstających. Rozkłady wartości cechy można porównywać w podgrupach; rozkład przedstawiany jest za pomocą tzw. „*pudełek*" (inaczej: „*skrzynka-wąsy*"); wybrane argumenty:
  - pierwszym jest wektor, lista wektorów, ramka danych lub formuła zawierające zmienne, które mają być zobrazowane; jeśli argumentem będzie wektor liczb, narysowane zostanie jedno „*pudełko*", a gdy będzie to więcej wektorów liczb – kilka „*pudełek*",
  - ***range*** – określa szerokość przedziału, poza którym obserwacje będą traktowane jako odstające,
  - ***outline*** – określa, czy mają zostać uwzględnione (**TRUE**) wartości odstające, czy też nie (**FALSE**),
  - ***varwidth*** – pozwala dostosowywać szerokość „*pudełek*" proporcjonalnie do liczby obserwacji,
- **wykres punktowy (kropkowy)** – funkcja **scatterplot()** z pakietu **car** (**car::scatterplot()**); obrazuje zależność pomiędzy dwiema zmiennymi liczbowymi, a także może być wzbogacona o wiele elementów; wybrane argumenty:
  - pierwszym są dwa wektory obejmujące zmienne, które mają zostać przedstawione na wykresie,
  - ***boxplots*** – definiuje, czy wzdłuż osi mają być rysowane wykresy pudełkowe,
  - ***reg.line*** – definiuje, czy na wykresie ma być dorysowana prosta regresji liniowej,
  - ***smooth*** – definiuje, czy na wykresie ma być dorysowana krzywa regresji wykładniczej,
- **macierz wykresów punktowych** – funkcja **pairs()**; dla każdej pary zmiennych numerycznych w ramce danych, jest rysowany wykres punktowy – dzięki temu można szybko sprawdzić zależności pomiędzy wszystkimi zmiennymi jednocześnie,
- **wykres kołowy** – funkcja **pie()** pozwala zwizualizować strukturę udziałów; wybrane argumenty to:
  - pierwszy stanowi wektor obejmujący wartości liczbowe, które mają być zobrazowane na wykresie,
  - ***main*** – tytuł wykresu,
  - ***labels*** – etykiety pozwalające na opis wycinków koła,
  - ***col*** – kolorystyka składowych wykresu,

- ***radius*** – promień okręgu,
- ***clockwise*** – zdefiniowanie, czy wykres ma być tworzony zgodnie z ruchem wskazówek zegara,

- **wykres mozaikowy** – funkcja **mosaicplot()** prezentuje relacje pomiędzy zmiennymi jakościowymi za pomocą prostokątów, których pole odpowiada liczebności danej kombinacji kategorii.

Przykład 9.2.

Wybrane wykresy w R.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **wykresy2.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
      #wykres słupkowy
values <- c(4, 6, 8)
labels <- c("A", "B", "C")
barplot(values, names.arg = labels, col = c("red", "green", "blue"),
        main = "Wykres słupkowy")

      #histogram
dane <- rnorm(100)   #100 losowych wartości z rozkładu normalnego
hist(dane, col = "lightblue", main = "Histogram", xlab = "Wartości")

      #wykres pudełkowy
data("mtcars")
boxplot(mpg ~ cyl, data = mtcars, col = "orange",
        main = "Zużycie paliwa wg liczby cylindrów")

      #wykres kołowy
pie(table(mtcars$cyl), col=rainbow(3), main = "Odsetek samochodów z różną
liczbą cylindrów")
sizes <- c(10, 20, 30)
labels <- c("A", "B", "C")
pie(sizes, labels = labels, col = rainbow(3), main = "Wykres kołowy")

      #wykres punktowy
library(car)
scatterplot(mpg ~ hp, data = mtcars, main = "Moc silnika, a spalanie")

      # macierz wykresów punktowych
pairs(mtcars[,c(1,3,4,7)], col = "blue", pch = 19)
```

```
      #wykres mozaikowy
mosaicplot(table(mtcars$cyl, mtcars$gear),
           main = "Liczba cylindrów, a liczba biegów",
           xlab = "Liczba cylindrów",
           ylab = "Liczba biegów",
           color = TRUE)
```

9.3. Istota pakietu ggplot2

R jest wyposażony w wiele pakietów, które pozwalają na wizualizację danych. Chociaż bazowe funkcje graficzne w R są wygodne i szybkie do zastosowania, a wygenerowane wykresy są dość czytelne, to w przypadku bardziej złożonych wykresów często stosuje się pakiet **ggplot2** – jest on jednym z najpopularniejszych narzędzi do tego celu w R. Pakiet ten opiera się na pewnej założonej koncepcji graficznej – tj. systematycznym podejściu do tworzenia wykresów poprzez nakładanie na siebie kolejnych warstw. Podstawę stanowi funkcja **ggplot()**, w której wskazuje się dane wejściowe. Następnie kolejne elementy dodaje się operatorem **+**. Przykładowe funkcje to:

- **geom_point()** – rysuje punkty,
- **geom_line()** – rysuje linie,
- **geom_histogram()** – tworzy histogram,
- **geom_bar()** – tworzy wykres słupkowy,
- **geom_density()** – przedstawia rozkład gęstości,
- **geom_boxplot()** – tworzy wykres pudełkowy,
- **geom_text()** – umieszcza opisy tekstowe na wykresie,
- **aes()** – definiuje mapowanie zmiennych na osie i inne właściwości graficzne (np. kolor, kształt, wielkość),
- **facet_wrap()** – dzieli wykres na osobne panele dla każdej wyróżnionej pogrupy według wartości określonej zmiennej,
- **labs()** – pozwala dodać etykiety osi i tytuł wykresu,
- **ggtitle()** – umożliwia dodanie tytułu wykresu,
- **stat_summary()** – dodaje podsumowania statystyczne, np. średnią arytmetyczną,
- **theme()** – modyfikuje wygląd całego wykresu (np. czcionki, tło).

Dzięki niezwykłej elastyczności i spójności, pakiet **ggplot2** jest obecnie standardem w tworzeniu estetycznych i profesjonalnych wizualizacji w R – więcej na: https://ggplot2.tidyverse.org.

Przykład 9.3.

Wybrane wykresy w R – pakiet ggplot2.

Utwórz nowy skrypt w programie RStudio Desktop o nazwie **wykresy3.R**. Wprowadź do niego poniższy kod (bez komentarzy po znaku #). Kod ten należy następnie uruchamiać polecenie po poleceniu.

```
library(ggplot2)
data("mtcars")

     #wykres punktowy z ggplot2: zużycie paliwa, a masa pojazdu
ggplot(mtcars, aes(x = wt, y = mpg)) +    #dodaj opcję color = factor(cyl)
  geom_point(size = 3) +
  geom_smooth(method = "lm", se = FALSE) +
  labs(title = "Zużycie paliwa, a masa pojazdu",
       x = "Waga (1000 funtów)", y = "mpg")

     #wykres liniowy z gggplot2: średnie spalanie w grupach wg cylindrów
     #uwaga: spalanie wyrażone jest w MPG (Miles Per Gallon) - czym wyższa
     #wartość, tym więcej mil pokonanych na jednym galonie paliwa
ggplot(mtcars, aes(x = factor(cyl), y = mpg, group = 1)) +
  stat_summary(fun = mean, geom = "line") +
  stat_summary(fun = mean, geom = "point", size = 3, color = "red") +
  ggtitle("Średnie spalanie według liczby cylindrów") +
  theme(plot.title = element_text(hjust = 0.5, size = 16, face = "bold"),
    axis.title.x = element_text(size = 12),
    axis.title.y = element_text(size = 12),
    panel.background = element_rect(fill = "gray95"),
    panel.grid.major = element_line(color = "blue"),
    panel.grid.minor = element_line(color = "green"))

     #histogram z ggplot2
ggplot(mtcars, aes(x = mpg)) +
  geom_histogram(binwidth = 2, fill = "lightblue", color = "black") +
  labs(title = "Histogram zużycia paliwa")

     #wykres słupkowy z ggplot2: liczba samochodów według liczby biegów
ggplot(mtcars, aes(x = factor(gear))) +
  geom_bar(fill = "darkgreen") +
  ggtitle("Liczba samochodów według liczby biegów")
```

```
      #wykres pudełkowy z ggplot2: spalanie według liczby cylindrów
ggplot(mtcars, aes(x = factor(cyl), y = mpg, fill = factor(cyl))) +
  geom_boxplot() +
  ggtitle("Rozkład spalania w zależności od liczby cylindrów")

      #wykres gęstości z ggplot2: rozkład przyspieszenia według liczby
      #cylindrów
ggplot(mtcars, aes(x = qsec, fill = factor(cyl))) +
  geom_density(alpha = 0.6) +
  ggtitle("Rozkład przyspieszenia według liczby cylindrów")

      #ggplot2: mpg vs hp, rozdzielone według liczby cylindrów
ggplot(mtcars, aes(x = hp, y = mpg)) +
  geom_point(color = "darkorange") +
  facet_wrap(~ cyl) +
  ggtitle("Zużycie paliwa, a moc silnika (według liczby cylindrów)")
```

10. Case study – poszkodowani w wypadkach przy pracy

W poprzednich rozdziałach przedstawiony został szereg technik umożliwiających przetwarzanie i analizę danych w języku R wraz z praktycznymi przykładami. Teraz nadszedł czas, aby podsumować zdobytą wiedzę przy wykorzystaniu spójnego przykładu praktycznego, a także poznać kilka nowych pakietów i technik. Ćwiczenie to pokaże pełny cykl analizy w języku R – od pozyskania danych z zewnętrznego źródła, przez ich przekształcenie i obliczenia pomocnicze, aż po utworzenie wizualizacji.

Niniejsze studium przypadku polega na analizie danych o poszkodowanych w wypadkach przy pracy w Polsce w latach 2010-2023, z podziałem na ogółem, kobiety i mężczyzn. Wykorzystane zostaną dane udostępniane przez Główny Urząd Statystyczny w ramach Banku Danych Lokalnych (“GUS - Bank Danych Lokalnych“, 2025) i dedykowanego do tego celu API oraz pakietu **bdl** (“The Comprehensive R Archive Network“, 2025).

Na wstępie należy załadować potrzebne pakiety, a w przypadku gdy są one niedostępne, należy je zainstalować:

```
if (!require(bdl)) {          #dostęp do API GUS – Bank Danych Lokalnych
  install.packages("bdl", dependencies = TRUE)
  library(bdl)
  } else {library(bdl)}

if (!require(tidyverse)) {    #zawiera dplyr, tidyr, ggplot2 itd.
  install.packages("tidyverse", dependencies = TRUE)
  library(tidyverse)
  } else {library(tidyverse)}

#w niektórych wykresach wykorzystana zostanie funkcja scale_y_continuous()
if (!require(scales)) {       #obejmuje funkcje skal do wizualizacji
  install.packages("scales", dependencies = TRUE)
  library(scales)
} else {library(scales)}

      #umożliwia „narysowanie” kilku wykresów na jednym
if (!require(gridExtra)) {
  install.packages("gridExtra", dependencies = TRUE)
  library(gridExtra)
} else {library(gridExtra)}
```

Następnie należy pobrać interesujące dane z API Banku Danych Lokalnych GUS przy wykorzystaniu pakietu **bdl** i zapoznać się z ich strukturą – rys. 14:

```
      #pobranie danych: poszkodowani w wypadkach przy pracy (ogółem,
      #kobiety, mężczyźni)
dane_bdl <- get_data_by_unit(
  unitId = "000000000000",   #cała Polska
  varId  = c("74034", "58355", "58357"),         #74034 - wypadki ogółem;
                                      #58355 - mężczyźni; 58357 - kobiety
  year   = 2010:2023,        #lata
  lang   = "pl"
)
str(dane_bdl)     #informacje o strukturze obiektu
head(dane_bdl)    #pierwsze 6 wierszy
```

```
> str(dane_bdl)
bdl [42 × 7] (S3: bdl/tbl_df/tbl/data.frame)
 $ id                  : chr [1:42] "58355" "58355" "58355" "58355" ...
 $ year                : chr [1:42] "2010" "2011" "2012" "2013" ...
 $ val                 : int [1:42] 64551 65814 60614 56170 55933 54979 54770 54995 52993 51478 ...
 $ measureUnitId       : int [1:42] 26 26 26 26 26 26 26 26 26 26 ...
 $ attrId              : int [1:42] 1 1 1 1 1 1 1 1 1 1 ...
 $ attributeDescription: chr [1:42] "" "" "" "" ...
 $ lastUpdate          : chr [1:42] "2024-11-22T09:20:14.423" "2024-11-22T09:20:14.423" "2024-11-22T09:20:14.423"
> head(dane_bdl)
# A tibble: 6 × 7
  id    year    val measureUnitId attrId attributeDescription lastUpdate
  <chr> <chr> <int>         <int>  <int> <chr>                <chr>
1 58355 2010  64551            26      1 ""                   2024-11-22T09:20:14.423
2 58355 2011  65814            26      1 ""                   2024-11-22T09:20:14.423
3 58355 2012  60614            26      1 ""                   2024-11-22T09:20:14.423
4 58355 2013  56170            26      1 ""                   2024-11-22T09:20:14.423
5 58355 2014  55933            26      1 ""                   2024-11-22T09:20:14.423
6 58355 2015  54979            26      1 ""                   2024-11-22T09:20:14.423
```

Rys. 14. Struktura i zawartość pobranego zbioru danych – dane_bdl

[Tekst alternatywny. Rysunek 14 przedstawia fragment okna programu RStudio Desktop. W konsoli pokazano wynik działania funkcji str() (struktura danych) oraz head() (6 pierwszych wierszy) dla ramki danych typu tibble o nazwie dane_bdl – zbioru pobranego z Banku Danych Lokalnych GUS.]

W kolejnym kroku należy wybrać interesujące kolumny oraz zamienić kody zmiennych na czytelne nazwy, uzyskując ramkę danych typu **tibble** o nazwie ***dane***, której fragment widoczny jest na rys. 15:

```
      #podstawowe przekształcenie: wybór kolumn i zmiana ich nazw
dane <- dane_bdl %>%
      select(rok = year, zmienna_id = id, poszkodowani = val)
```

```
     #zamiana identyfikatorów zmiennych na czytelne nazwy
dane$zmienna_id <- ifelse(dane$zmienna_id == "58355", "Wm",
     dane$zmienna_id)
dane$zmienna_id <- ifelse(dane$zmienna_id == "58357", "Wk",
     dane$zmienna_id)
dane$zmienna_id <- ifelse(dane$zmienna_id == "74034", "W",
     dane$zmienna_id)
head(dane)
```

```
> head(dane)
# A tibble: 6 × 3
  rok   zmienna_id poszkodowani
  <chr> <chr>              <int>
1 2010  Wm                 64551
2 2011  Wm                 65814
3 2012  Wm                 60614
4 2013  Wm                 56170
5 2014  Wm                 55933
6 2015  Wm                 54979
```

Rys. 15. Fragment ramki danych typu tibble o nazwie dane

[Tekst alternatywny. Rysunek 15 przedstawia fragment okna programu RStudio Desktop. W konsoli pokazano wynik działania funkcji head() (6 pierwszych wierszy) dla ramki danych typu tibble o nazwie dane, po wykonaniu operacji z powyższych fragmentów kodu.]

Poniżej dokonano przekształcenia danych do odpowiedniego formatu – przeprowadzono reorganizację ich formatu na „szeroki”, oraz konwersję zmiennej ***rok*** na typ numeryczny, uzyskując ramkę danych typu **tibble** o nazwie ***dane_wide***, której fragment widoczny jest na rys. 16:

```
     #przekształcenie do formatu szerokiego; kolumny: rok, W, Wk, Wm
dane_wide <- dane %>% pivot_wider(names_from = zmienna_id,
     values_from = poszkodowani)

     #konwersja zmiennej rok na typ numeryczny
dane_wide$rok <- as.numeric(dane_wide$rok)
head(dane_wide)
```

```
> head(dane_wide)
# A tibble: 6 × 4
  rok      Wm    Wk     W
  <chr> <int> <int> <int>
1 2010  64551 29656 94207
2 2011  65814 31408 97222
3 2012  60614 30386 91000
4 2013  56170 32097 88267
5 2014  55933 32708 88641
6 2015  54979 32643 87622
```

Rys. 16. Fragment ramki danych typu tibble o nazwie dane_wide

[Tekst alternatywny. Rysunek 16 przedstawia fragment okna programu RStudio Desktop. W konsoli pokazano wynik działania funkcji head() (6 pierwszych wierszy) dla ramki danych typu tibble o nazwie dane_wide, po wykonaniu operacji z powyższych fragmentów kodu.]

Następnie obliczono dodatkowe wskaźniki: udziały procentowe poszkodowanych kobiety i mężczyzn w ogólnej liczbie poszkodowanych w wypadkach przy pracy, w wyniku czego dodano dwie nowe kolumny do ramki danych ***dane_wide*** – rys. 17:

```
     #dodanie nowych kolumn obliczeniowych: udział kobiet i mężczyzn w %
dane_wide <- dane_wide %>%
     mutate(
        Odsetek_kobiet = Wk / W * 100,
        Odsetek_mezczyzn = Wm / W * 100)
head(dane_wide)
```

```
> head(dane_wide)
# A tibble: 6 × 6
    rok    Wm    Wk     W Odsetek_kobiet Odsetek_mezczyzn
  <dbl> <int> <int> <int>          <dbl>             <dbl>
1  2010 64551 29656 94207           31.5              68.5
2  2011 65814 31408 97222           32.3              67.7
3  2012 60614 30386 91000           33.4              66.6
4  2013 56170 32097 88267           36.4              63.6
5  2014 55933 32708 88641           36.9              63.1
6  2015 54979 32643 87622           37.3              62.7
```

Rys. 17. Fragment ramki danych typu tibble o nazwie dane_wide

[Tekst alternatywny. Rysunek 17 przedstawia fragment okna programu RStudio Desktop. W konsoli pokazano wynik działania funkcji head() (6 pierwszych wierszy)

dla ramki danych typu tibble o nazwie dane_wide, po wykonaniu operacji z powyższych fragmentów kodu; widoczne są dwie nowe kolumny.]

Na koniec, utworzono 6 przykładowych wykresów przy wykorzystaniu pakietu **ggplot2** i „narysowano" je na jednym, wspólnym wykresie przy wykorzystaniu pakietu **gridExtra** – rys. 18:

```
      #wykres liniowy – poszkodowani w wypadkach przy pracy ogółem
w1 <- ggplot(dane_wide, aes(x = rok, y = W)) +
     geom_line(color = "blue", linewidth = 1) +
     geom_point(color = "darkblue") +
     labs(
        title = "Poszkodowani w wypadkach przy pracy w Polsce (ogółem)",
       x = "Rok", y = "Liczba poszkodowanych") +
     theme_minimal()

      #wykres słupkowy – liczba poszkodowanych ogółem
w2 <- ggplot(dane_wide, aes(x = factor(rok), y = W)) +
     geom_col(fill = "steelblue") +
     labs(
        title = "Liczba poszkodowanych w wypadkach przy pracy (ogółem)",
        x = "Rok", y = "Liczba poszkodowanych") +
     theme_minimal()

      #przekształcenie danych na potrzeby tworzonych wykresów
dane_long <- dane_wide %>% pivot_longer(cols = c("W", "Wm", "Wk"),
     names_to = "grupa", values_to = "liczba")

      #wykres słupkowy – kobiety, mężczyźni i ogółem (obok siebie)
w3 <- ggplot(dane_long, aes(x = factor(rok), y = liczba, fill = grupa)) +
     geom_col(position = "dodge") +
     labs(
        title = "Poszkodowani w wypadkach przy pracy w Polsce",
        x = "Rok", y = "Liczba poszkodowanych",
        fill = "Grupa") +
     theme_minimal()

      #wykres liniowy – kobiety, mężczyźni i ogółem
w4 <- ggplot(dane_long, aes(x = rok, y = liczba, color = grupa)) +
     geom_line(linewidth = 1) +
     geom_point(size = 2) +
     labs(
```

```
       title = "Poszkodowani w wypadkach przy pracy w Polsce (2010–2023)",
       x = "Rok", y = "Liczba poszkodowanych",
       color = "Grupa") + scale_y_continuous(labels = comma) +
     theme_minimal()

      #wykres kołowy – udział kobiet i mężczyzn w roku 2023
      #wyodrębnienie danych dla roku 2023
dane_2023 <- dane_wide %>% filter(rok == 2023) %>%
     select(Odsetek_kobiet, Odsetek_mezczyzn) %>%
     pivot_longer(everything(), names_to = "grupa", values_to = "udzial")

w5 <- ggplot(dane_2023, aes(x = "", y = udzial, fill = grupa)) +
     geom_col(width = 1) +
     coord_polar("y") +
     labs(
        title = "Udział kobiet i mężczyzn w wypadkach przy pracy (2023)",
        x = NULL, y = NULL, fill = "Grupa") +
  theme_void()

      #wykres pudełkowy – rozkład liczby poszkodowanych wg grup
w6 <- ggplot(dane_long, aes(x = grupa, y = liczba, fill = grupa)) +
     geom_boxplot(alpha = 0.7) +
     labs(
        title = "Poszkodowani (2010–2023)",
        x = "Grupa", y = "Liczba poszkodowanych", fill = "Grupa") +
     scale_y_continuous(labels = comma) +
     theme_minimal()

      #6 wykresów na jednym – pakiet gridExtra
grid.arrange(w1, w2, w3, w4, w5, w6, nrow = 3, ncol = 2)
```

[Tekst alternatywny. Rysunek 18 przedstawia zestaw sześciu wykresów (różnych typów: słupkowe, liniowe, kołowy i pudełkowy) utworzonych w języku R, przy wykorzystaniu pakietu ggplot2. Pokazano m.in. liczbę poszkodowanych w wypadkach przy pracy ogółem.]

Rys. 18. Wizualizacja danych – sześć przykładowych wykresów

Zakończenie

W niniejszych materiałach dydaktycznych do przedmiotu ***Programowanie i analiza danych w R*** omówiono najważniejsze zagadnienia przydatne podczas pracy w języku R – od podstawowych struktur danych i składni, przez wybrane sposoby ich importu, transformacji i analizy, aż po przykładowe metody wizualizacji wyników. Skupiono się na elementach najistotniejszych z perspektywy studiów na kierunku ***inżynieria danych*** – zarówno w zakresie wiedzy teoretycznej, jak też przykładach praktycznych. Informacje zawarte w tych materiałach stanowią zarówno poszerzenie treści podstawowych realizowanych w trakcie trwania kursu, jak też pozwalają Studentowi na rozszerzenie wybranych umiejętności w kontekście przyszłej pracy z danymi – wiele zagadnień z niniejszego opracowania wykracza poza ramy kursu.

W ogólności należy mieć jednak świadomość, że język R to potężne narzędzie, które oferuje znacznie szersze możliwości w zakresie analizy danych, posiadając różnorodne i pokaźne pakiety tematyczne – obejmujące m.in. rozbudowane modelowanie statystyczne, zaawansowane techniki uczenia maszynowego, czy też integrację z innymi środowiskami i językami programowania. Niniejsze materiały mają zatem charakter wstępny i stanowią swoisty przewodnik po podstawach, natomiast zdobyta wiedza powinna być punktem wyjścia do dalszego, samodzielnego rozwijania kompetencji w zakresie programowania i analizy danych w języku R. Tematy bardziej zaawansowane wymagają osobnych, bardziej pogłębionych opracowań.

Z czasem Student, sięgając po dokumentację, dostępną literaturę i kolejne pakiety, będzie mógł stopniowo poszerzać swoje umiejętności, aby wykorzystywać język R w coraz bardziej wymagających projektach.

Politechnika Świętokrzyska
Kielce University of Technology
Projekt „Dostosowanie kształcenia w Politechnice Świętokrzyskiej do potrzeb współczesnej gospodarki”
nr FERS.01.05-IP.08-0234/23

Bibliografia

[1]. Biecek P. (2016). *Odkrywać! Ujawniać! Objaśniać! Zbiór esejów o sztuce prezentowania danych*, Fundacja Naukowa SmarterPoland.pl, Warszawa.
[2]. Biecek P. (2017). *Przewodnik po pakiecie* R, wydanie IV, Oficyna Wydawnicza GiS, Wrocław.
[3]. Character encoding. (2025-08-08). W: *Wikipedia*, (https://en.wikipedia.org/w/index.php?title=Character_encoding&oldid=1304914746, dostępny 22.08.2025).
[4]. Gągolewski M. (2014). *Programowanie w języku R. Analiza danych, obliczenia, symulacje*, Wydawnictwo Naukowe PWN SA, Warszawa.
[5]. GUS - Bank Danych Lokalnych. (2025). (https://bdl.stat.gov.pl, dostępny 23.08.2025).
[6]. Lander J. P. (2018). *R dla każdego. Zaawansowane analizy i grafika statystyczna*, APN Promise, Warszawa.
[7]. Medeiros K. (2018). *R Programming Fundamentals. Deal with data using various modeling techniques*, Packt Publishing Ltd., Birmingham, UK.
[8]. Mount G. (2022). *Zaawansowana analiza danych. Jak przejść z arkuszy Excela do Pythona i R*, Helion, Gliwice.
[9]. Nowosad J. (2020). *Elementarz programisty: wstęp do programowania używając R*, Space A, Poznań, (https://nowosad.github.io/elp/, dostępny 02.07.2025).
[10]. Nwanganga F., & Chapple M. (2022). *Praktyczne uczenie maszynowe w języku R*, Wydawnictwo APN PROMISE S.A., Warszawa.
[11]. R (programming language). (2025-06-16). W: *Wikipedia*, (https://en.wikipedia.org/w/index.php?title=R_(programming_language)&oldid=1295926117, dostępny 17.06.2025).
[12]. Rdocumentation. (2025). (https://www.rdocumentation.org/, dostępny 17.06.2025) – wraz z podstronami.
[13]. Regular expression. (2025-08-13). W: *Wikipedia*, (https://en.wikipedia.org/w/index.php?title=Regular_expression&oldid=1305664141, dostępny 22.08.2025).
[14]. RStudio Desktop. (2025). (https://posit.co/download/rstudio-desktop/, dostępny 17.06.2025).
[15]. RStudio. (2025-03-24). W: *Wikipedia*, (https://en.wikipedia.org/w/index.php?title=RStudio&oldid=1282174211, dostępny 17.06.2025).

[16]. S (programming language). (2025-02-18). W: *Wikipedia*, (https://en.wikipedia.org/w/index.php?title=S_(programming_language)&oldid=1276388529, dostępny 17.06.2025).
[17]. Scraping Data. (2025). W: *AFIT Data Science Lab R Programming Guide*, (https://afit-r.github.io/scraping, dostępny 17.06.2025).
[18]. The Comprehensive R Archive Network. (2025). (https://cran.r-project.org, dostępny 17.06.2025) – wraz z podstronami.
[19]. The R Foundation (2025). *The R Project for Statistical Computing*, (https://www.r-project.org/, dostępny 17.06.2025) – wraz z podstronami.
[20]. Tidyverse. (2025). (https://www.tidyverse.org/, dostępny 17.06.2025) – wraz z podstronami.
[21]. Walesiak M., & Gatnar E. (red. nauk.). (2012). *Statystyczna analiza danych z wykorzystaniem programu R*, Wydawnictwo Naukowe PWN, Warszawa.