

# **Pogromcy Danych Przetwarzanie danych w programie R**

**Przemysław Biecek  
Uniwersytet Warszawski**

Ten plik zawiera odcinki z pierwszego sezonu Pogromców Danych. Wszystkie zamieszczone tutaj materiały są dostępne na stronie <http://pogromcydanych.icm.edu.pl>

Materiały dostępne bezpłatnie. Materiały można wykorzystywać wyłącznie do celów edukacyjnych.

Kurs został wykonany w ramach projektu WSAD, współfinansowanego przez Unię Europejską w ramach Europejskiego Funduszu Społecznego.

# Intro dla kursu „pogRomcy danych”

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 1*  
*pogRomcy danych*

- [Co to za kursy?](#)
- [Pierwszy sezon jest o programie R](#)
- [Drugi sezon jest o wizualizacji i modelowaniu](#)
- [Stromej krzywej uczenia nie należy się obawiać](#)

## Co to za kursy?

*PogRomcy danych* to zbiór dwóch kursów o przetwarzaniu, wizualizacji i modelowaniu danych.

Kursy (nazywane też sezonami) zostały przygotowane dla osób, które pracują, chciałyby pracować, lub są ciekawe jak wygląda, praca z danymi. Oba kursy są prowadzone na poziomie podstawowym. Zakładamy jedynie ogólną orientację jak wygląda tabela z danymi w arkuszach

kalkulacyjnych, np. takich jak Excel, oraz ogólną umiejętność pracy z komputerem.

Wcześniejsza znajomość programowania będzie bardzo przydatna, ale nie jest konieczna. Wszystkie nowe pojęcia pokazujemy na przykładach związanych z analizą ciekawych danych.

W pierwszym kursie pokażemy jak pracować z programem R, bezpłatnym i bardzo potężnym narzędziem do analizy danych. Pokażemy jak pisać programy, tworzyć funkcje, pętle, wczytywać, przetwarzać i zapisywać dane.

W drugim kursie pokażemy co wyróżnia zły, a co dobry wykres. Następnie pokażemy jak tworzyć dobre wykresy korzystając z programu R. Druga część dotyczy modeli i metod statystycznych. Pokażemy jak badać zależności pomiędzy dwoma lub większą liczbą zmiennych.

Oba kursy zwieńczone są zbiorem 20 zadań do samodzielnego wykonania. Osoby, które wykonają 13 lub więcej zadań otrzymają tytuł „Pogromców danych” oraz certyfikat ukończenia kursu sygnowany przez Uniwersytet Warszawski.

Aby uzyskać dostęp do materiałów potrzebna jest rejestracja. Jest ona bezpłatna i otwarta dla każdego zainteresowanego. Zarejestrowani uzyskują dostęp do

materiałów dydaktycznych, forum oraz zadań. Materiały dydaktyczne zostały podzielone na krótkie bloki, omawiające kolejne zagadnienia. Każdy blok kończy się krótkimi sprawdzającymi zadaniami. W materiałach znajdują się również przykładowe odpowiedzi do tych zadań.

Do obu kursów dostępne jest forum, ułatwiające komunikację pomiędzy uczestnikami. Jeżeli jakaś porcja materiału jest trudna, sprawia kłopoty, lub jeżeli w trakcie pracy pojawią się ciekawe rozwiązania, warto podzielić się nimi na forum. Obsługa kursu śledzi forum i odpowiada na wszelkie, nawet najbardziej podstawowe pytania. Również inni uczestnicy kursu mogą coś zasugerować lub pomóc w rozwiązaniu problemu.

---

## **Pierwszy sezon jest o programie R**

Pierwszy kurs, jest poświęcony programowi R. Rozpoczyna się od czterech odcinków pokazujących dlaczego warto nauczyć się tego programu, jak go zainstalować, jak doinstalować wygodny edytor oraz dodatkowe pakiety do pracy z R. Następnie omawiane są podstawy pracy z programem R. Pokażemy jak i gdzie wpisywać komendy oraz gdzie spodziewać się wyników.

Kolejne trzy odcinki poświęcone są wczytywaniu i zapisywaniu tabel z danymi, oraz podstawowym operacjom na tych tabelach z danymi. Pokażemy jak wczytać plik tekstowy lub w formacie Excela do programu R. Jak zapisać wyniki oraz jak z wyników wybrać tylko określone kolumny lub wiersze.

Odcinki 8, 9, 10 pokazują jak automatyzować obliczenia. Wyjaśniamy w nich na przykładach jak działają pętle, instrukcje warunkowe oraz jak tworzyć własne funkcje.

Odcinki od 11 do 15 wprowadzają funkcje i operacje typowe dla najpopularniejszych rodzajów. Pokażemy co można zrobić z wektorem liczb, a co z wektorem napisów lub dat. Wyjaśnimy czym są zmienne logiczne i jakościowe oraz jakie operacje na nich można wykonać.

Odcinki od 16 do 24 przedstawiają szeroki wachlarz funkcji pozwalających na swobodne przekształcanie danych. Opanowawszy te funkcje będziemy potrafili wykonać 90% typowych prac związanych z przetwarzaniem danych. Będziemy wiedzieć jak filtrować, przekształcać, agregować, grupować dane i zmieniać ich strukturę.

Ten kurs jest kursem podstawowym, ale kończy go odcinek pokazujący gdzie można szukać dalszych informacji o programie R. Jeżeli program R przypadnie

nam do gustu, w ostatnim odcinku pokażemy jak go dalej zgłębiać.

---

## **Drugi sezon jest o wizualizacji i modelowaniu**

Pierwszy kurs pokazuje jak przetwarzać dane, drugi kurs pokazuje po co to robić. W tej części przedstawimy zagadnienia związane z eksploracją danych, wizualizacją oraz modelowaniem statystycznym.

Rozpoczynamy od trzech odcinków pokazujących kamienie milowe w historii wizualizacji danych, trudności związane z prezentowaniem danych, problemy wynikające ze sposobu w jaki nasz mózg interpretuje obraz a następnie pokażemy przykłady rzetelnych oraz nierzetelnych wykresów.

Odcinki 4, 5, 6 i 7 wprowadzą nas w świat pakietu ggplot2. Poznamy podstawowe i bardziej zaawansowane instrukcje, pozwalające na wykonanie praktycznie dowolnego wykresu w programie R. Jeżeli praca z wykresami w programie R nas zacieka, to w odcinku 8 przedstawimy sugestie dalszych materiałów rozwijających nasze umiejętności wizualizacji danych.

Kolejne odcinki przedstawiają podstawowe narzędzia statystyczne służące do eksploracji danych do analizy statystycznej danych. Pokażemy jak statystycznie porównać dwie średnie. Jak sprawdzić czy dwa zjawiska występują niezależnie czy też czy jest pomiędzy nimi jakaś zależność. Oraz pokażemy jak tworzyć modele regresyjne np. aby prognozować pewną cechę w przyszłości. Również ten blok zakończy się listą propozycji literaturowych w których można znaleźć więcej informacji na temat modelowania statystycznego.

Do obu kursów dodaliśmy przykłady bardzo ciekawych ale i nietypowych zastosowań analizy danych. Pokażemy jak analiza danych jest wykorzystywana w analizie danych sondażowych, robotyce, inżynierii lingwistycznej i meteorologii.

---

## **Stromej krzywej uczenia nie należy się obawiać**

Analiza danych, a w szczególności analiza z użyciem programu R, charakteryzuje się stromą krzywą uczenia. Na początku wiele rzeczy będzie nowych i trudnych. Przygotowaliśmy jednak wiele materiałów i na bieżąco monitorujemy forum, aby odpowiadać na wszelkie pytania i pomagać w stawianiu pierwszych kroków.



Gwarantujemy jednak, że wysiłek włożony w poznawanie programu R opłaci się. Lepiej będziemy mogli zrozumieć co właściwie robimy w analizach. Z czasem będziemy coraz sprawniej przetwarzać dane, a elastyczność i ekspresja programu R powodują, że praktycznie nie będzie przed nami barier związanych z analizą najróżniejszych danych.

Pracując ze slajdami, możemy w każdej chwili nacisnąć przycisk **T**, rozwinię on spis treści lub przycisk **A**, aby zamienić slajdy na ciągły dokument.

# Dlaczego R?

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 2*

*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Dlaczego R?](#)
- [Dlaczego R?](#)
- [Dlaczego R?](#)
- [Czy warto uczyć się programować, czy nie lepiej wybrać klikanie?](#)
- [A co o tym myślą inni?](#)

## O czym jest ten odcinek

Jest wiele powodów, dla których warto poznać język R do analizy danych.

W tym odcinku pokażemy dlaczego warto poznać R, co można w tym języku zrobić oraz powiemy jak się tego języka nauczyć. Powiemy też jakie umiejętności są potrzebne by efektywnie pracować z danymi w tym czy

innym narzędziu oraz powiemy jak te umiejętności nabyć.

Czy ten język i ten kurs jest dla Ciebie?

Być może jesteś zaawansowanym w boju badaczem danych z doskonałą znajomością SPSS / SAS / Statistica czy Stata.

Być może jesteś analitykiem biznesowym z niezłą znajomością Excela lub Tableau.

Być może Twoje dotychczasowe doświadczenia to programowanie w językach takich jak Python, Java czy C++.

Być może masz za sobą pierwsze przymiarki do R czy to na zajęciach czy w ramach kształcenia się.

A może jeszcze nie trafiłeś na rynek pracy i z czysto poznawczej ciekawości zajrzałeś na stronę tego kursu.

R jest dla Ciebie. Tak czy inaczej aby analizować dane potrzebujesz dobrego narzędzia i umiejętności pracy z danymi.

Ten odcinek ma na celu zachęcenie Cię do dokładniejszego poznania języka R.

---

# Dlaczego R?

*R jest dojrzałym językiem programowania zaprojektowanym z myślą o analizie danych oraz wizualizacji danych dostępnym bezpłatnie na otwartej licencji GPL.*

Przyjrzyjmy się temu zdaniu dokładniej.

*R jest językiem programowania.* Oznacza to, że nie jest ograniczony do kilku algorytmów, które przewidzieli twórcy, ale każdy może napisać w nim własny algorytm. Co więcej, wiele osób korzysta z tej opcji i tworzy nowe algorytmy, które inni użytkownicy R mogą wykorzystywać. Dzięki temu liczba algorytmów i funkcji dostępnych w R bardzo szybko się zwiększa.

*Jest dojrzałym językiem.* Jest rozwijany od ponad 21 lat, dzięki czemu zdążył nabrać masy krytycznej. Pewne usterki projektowe wczesnych wersji R (np. ograniczenie do 4GB RAM) zostały dostrzeżone i wyeliminowane. Dziś jest to narzędzie rozwijane i przez dużą grupę statystyków, i inżynierów oprogramowania, co jest gwarancją stabilnego rozwoju.

*Zaprojektowanym z myślą o analizie danych.* Dojrzałych języków programowania jest wiele, ale niewiele z nich

nadaje się do interaktywnej pracy z danymi. W języku R połączono wybrane cechy języków funkcyjnych oraz obiektowych, a nacisk położono na pracę interaktywną z danymi. Dodatkowe biblioteki wspierają łatwe tworzenie raportów z wynikami. Dzięki tym cechom R jest stworzony do analizy danych.

*Wizualizacji danych.* W języku R można wykonać grafiki statystyczne o publikacyjnej jakości. Oznacza to, że bez dodatkowych narzędzi można stworzyć profesjonalny wykres. Co więcej, jest wiele pakietów, dzięki czemu tworzenie takich wykresów jest proste. Omówimy je w drugim sezonie tego kursu.

*Dostępny bezpłatnie na otwartej licencji [GPL](#).* Program R jest dostępny bezpłatnie do każdych zastosowań. Czy to na uczelni, czy w działalności komercyjnej, możemy go wykorzystywać bez żadnych opłat. Program R, jest dostępny na otwartej licencji, co oznacza, że każdy ma dostęp do źródeł, każdy element można zobaczyć jak funkcjonuje, można sprawdzić czy nie zawiera błędów i ewentualnie usprawnić.

---

## Dlaczego R?

Początkowo R zyskał popularność na uczelniach. Był i jest

wykorzystywany do nauczania analizy danych jak i do prowadzenia badań naukowych. Można śmiało powiedzieć, że obecnie R jest głównym narzędziem używanym w dydaktyce na dobrych uczelniach. Zdominował prowadzenie badań naukowych w wielu dziedzinach, takich jak bioinformatyka czy genetyka. Jest bardzo popularny w zastosowaniach medycznych, finansowych i wielu innych.

Od kilkunastu lat rośnie popularność R w przemyśle. Jedną z przyczyn jest to, że jest to narzędzie bezpłatne. Jest w nim również dostępnych bardzo wiele funkcji. Ale popularność swoją zawdzięcza też temu, że coraz więcej analityków zostało “wykształconych na programie R” oraz ten program zna. Rośnie liczba ofert pracy dla osób znających język R.

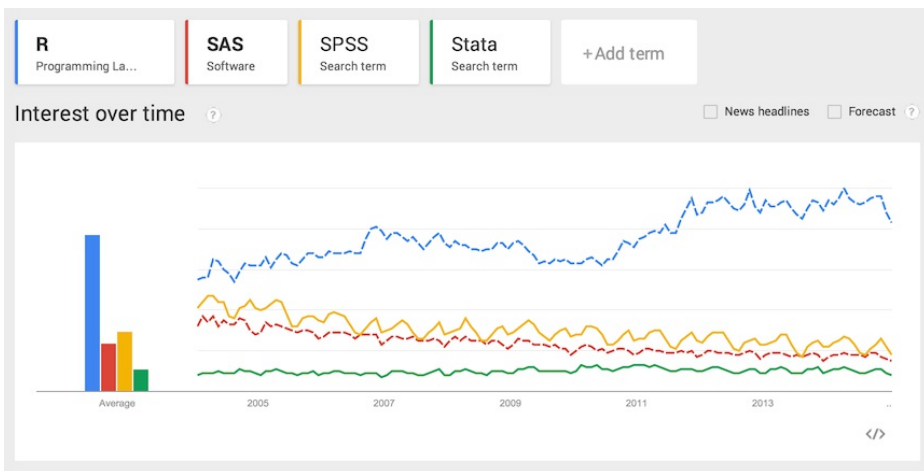
Małe firmy często wybierają R by ciąć koszty, ale R jest również używany przez gigantów. Zarówno przez firmy analityczne (jako silnik analityczny), jak i przez media. Jest używany w Google, Facebook, NY Times, New Scientist i wielu innych firmach.

R doskonale też integruje się z innymi rozwiązaniami informatycznymi, takimi jak Python, Java, C, C++, Hadoop, bazy danych.

---

# Dlaczego R?

Zgodnie z wynikami Google Trends, zainteresowanie R rośnie (wzrosło dwukrotnie w czasie ostatnich dziesięciu lat), podczas gdy zainteresowanie innymi pakietami do analizy danych, takimi jak SPSS, SAS czy Stata maleje.



## Czy warto uczyć się programować, czy nie lepiej wybrać klikanie?

Praca w R wymaga umiejętności programowania. Dla osób przyzwyczajonych do klikanych narzędzi oznacza to trudne początki, gdy muszą porzucić stare i złe nawyki oraz nauczyć się składni i nowych nazw funkcji. Jest to

trudne i wymaga czasu, ale warto przynajmniej z kilku powodów.

Jeżeli raz opracujemy skrypt wykonujący pewną pracę, możemy następnie ją łatwo powtórzyć. Wystarczy cały skrypt skopiować. Dzięki temu w miarę nabywania doświadczenia zwiększamy nasz potencjał.

Mamy zapisane komendy, które użyliśmy do analiz. Jeżeli po roku chcemy je odtworzyć jest to prostsze niż w przypadku klikanych narzędzi.

Jeżeli otrzymujemy dziwne wyniki, to całą ścieżkę analizy, która prowadzi do tych wyników możemy przeanalizować. Jest ona zapisana krok po kroku jako skrypt. Nie musimy jej przywoływać z pamięci, mogą z tych skryptów skorzystać też inne osoby.

---

## **A co o tym myślą inni?**

Moje przekonanie, że R jest doskonałym narzędziem do analizy danych utrzymuje się od kilkunastu lat. Osiem lat temu napisałem „Przewodnik po pakiecie R”, dostępny na stronie <http://biecek.pl/R>, który do dzisiaj doczekał się trzeciego wydania. Nie jestem z pewnością bezstronnym obserwatorem, dlatego warto przyjrzeć się temu co o R



mówią analitycy z firm, szpitali czy uczelni.

Bardzo ciekawe zestawienie R z innymi pakietami do analizy danych (SAS/SPSS/Stata/Statistica) znajduje się pod tym adresem [http://www.burns-stat.com/pages/Tutor/R\\_relative\\_statpack.pdf](http://www.burns-stat.com/pages/Tutor/R_relative_statpack.pdf).

Poniżej zamieszczam kilka cytatów analityków pracujących na co dzień z danymi, a którzy docenili R.

*One of the greatest advantages of R: getting your work done better and in less time.*

Frank Harrell, Biostatistics, Vanderbilt University

*R has really become the second language for people coming out of grad school now, and there's an amazing amount of code being written for it,*

Max Kuhn, associate director of nonclinical statistics at Pfizer.

*R is really important to the point that it's hard to overvalue it. It allows statisticians to do very intricate and complicated analyses without knowing the blood and guts of computing systems.*

Google research scientist, quoted in the New York Times

*The great beauty of R is that you can modify it to do all sorts of things, and you have a lot of prepackaged stuff that's already available, so you're standing on the shoulders of giants.*

Google chief economist, quoted in the New York Times

# Jak zainstalować R, RStudio oraz dodatkowe pakiety?

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 3  
pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Instalacja R](#)
- [Instalacja R](#)
- [Instalacja RStudio Desktop](#)
- [Instalacja RStudio Desktop](#)
- [Instalacja dodatkowych pakietów](#)
- [Gotowi do pracy](#)

## O czym jest ten odcinek

Aby wygodnie pracować z programem R potrzebujemy zainstalować silnik obliczeniowy R, edytor dzięki któremu praca z R będzie bardziej wygodna, taki jak np. RStudio,

oraz dodatkowe biblioteki z danymi i funkcjami. Wszystkie te elementy można pobrać z Internetu bezpłatnie.

W tym odcinku pokażemy:

- Jak zainstalować program R?
- Jak zainstalować program RStudio?
- Jak zainstalować dodatkowe biblioteki / pakiety do programu R?

Do instalacji tych elementów potrzebny jest dostęp do Internetu.

---

## Instalacja R

Najnowszą wersję programu R (na dzień dzisiejszy jest to wersja 3.1.3) można pobrać ze strony <http://cran.r-project.org/>. Nie musimy pamiętać tego adresu, wystarczy wpisać w Google ‘R’ a powyższy adres będzie pierwszym linkiem.

Na stronie w zakładce *Download* znaleźć można źródła do pobrania oraz program binarny działający na popularnych systemach operacyjnych (Windows / OSX / Linux). Instalacja sprowadza się do klikania *dalej, dalej*.



CRAN

[Mirrors](#)[What's new?](#)[Task Views](#)[Search](#)[About R](#)[R Homepage](#)[The R Journal](#)[Software](#)[R Sources](#)[R Binaries](#)[Packages](#)[Other](#)[Documentation](#)[Manuals](#)[FAQs](#)[Contributed](#)

## The Comprehensive R Archive Network

## Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

## Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2014-10-31, Pumpkin Helmet) [R-3.1.2.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#).

# Instalacja R

Program R można bez problemu zainstalować i używać na systemach operacyjnych Windows, OSX, większości dystrybucji Linuxa.

Program R możemy zainstalować również na przenośnym nośniku USB, dzięki czemu będziemy mogli uruchamiać to środowisko na różnych komputerach.

Nie potrzebujemy specjalnych uprawnień administracyjnych aby zainstalować program R. Jeżeli korzystamy z komputera do którego nie mamy pełni

dostępny, można zainstalować program R w swoim lokalnym katalogu.

Program R jest całkowicie bezpłatny do wszelkich zastosowań, edukacyjnych, przemysłowych, biznesowych, komercyjnych i hobbystycznych. Program R jest dostępny na licencji GPL 2.

Co roku oddawana jest do użycia nowa wersja R, tak więc w trakcie kursu może się okazać, że wersja 3.2 stanie się oficjalną (teraz jest tzw. wersją deweloperską). Zdecydowana większość funkcji działa identycznie pomiędzy wersjami, więc nie musimy obawiać się, że coś nie działa, jeżeli zainstalowaną mamy starszą wersję R. Wersja 3.1.3 lub nowsza wystarczy aby zrealizować ten kurs.

Praca z programem R jest interaktywna, co oznacza, że wpisując polecenia do konsoli R otrzymujemy natychmiast wyniki wykonanych poleceń.

---

## **Instalacja RStudio Desktop**


Najnowszą wersję RStudio Desktop (na dzień dzisiejszy jest to wersja 0.98) można pobrać ze strony

<http://www.rstudio.com/products/rstudio/download/>.

Nie musimy pamiętać tego adresu, wystarczy wpisać w Google ‘R Studio download’ a powyższy adres będzie pierwszym linkiem.

RStudio jest komercyjnym produktem rozwijanym przez firmę RStudio. Ten program jest dostępny bezpłatnie do większości zastosowań na licencji AGPL 3. Jego bardziej rozbudowana wersja, z dodatkowymi możliwościami jest dostępna odpłatnie.

www.rstudio.com/products/rstudio/download/

ProductsResourcesPricingAbout UsBlog

Download RStudio

Home / Overview / RStudio / Download RStudio

RStudio is a set of integrated tools designed to help you be more productive with R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.

If you run R on a Linux server and want to enable users to remotely access RStudio using a web browser [please download RStudio Server](#).

**Do you need support or a commercial license?**  
[Check out our commercial offerings](#)

**Click here to learn more about Shiny!**

**Download RStudio Desktop v0.98.1091** — [Release Notes](#)

RStudio requires R 2.11.1 (or higher). If you don't already have R, you can download it [here](#).

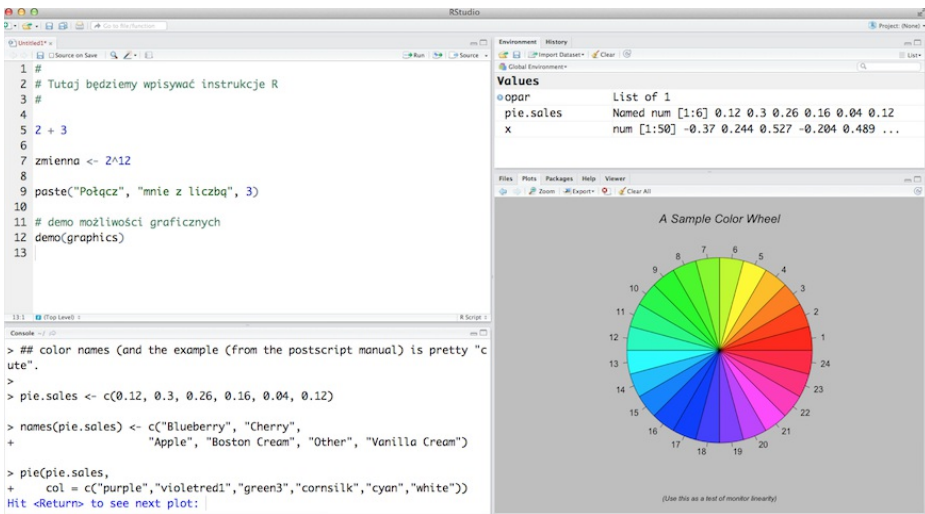
**Installers for ALL Platforms**

## Instalacja RStudio Desktop

Instalacja sprawdza się do klikania *dalej, dalej*. Po zainstalowaniu należy uruchomić program RStudio.

- Przy domyślnych ustawieniach, w programie RStudio wyświetlane są cztery panele. Lewy górny róg to miejsce w którym pisze się instrukcje. Te instrukcje można zaznaczyć i wysłać do wykonania poleceniem CTRL+Enter.
- Lewy dolny panel to konsola programu R, tutaj wyświetlane są instrukcje wprowadzone do R oraz ich wyniki.
- Prawy dolny panel przedstawia wykresy wyprodukowane przez instrukcje w R. W tym panelu jest też wyświetlana pomoc dla funkcji.
- Prawy górny panel pokazuje jakie obiekty znajdują się obecnie w pamięci R oraz jak są one duże. Klikając na wybrane obiekty możemy podejrzeć ich zawartość. Każdy wiersz to obiekt. Ale po pierwszym uruchomieniu to okno będzie puste, ponieważ nic jeszcze nie wczytaliśmy do pamięci.





# Instalacja dodatkowych pakietów

Program R po instalacji posiada setki przydatnych funkcji. Ale prawdziwa siła kryje się w dodatkowych pakietach, które można w każdej chwili dodać. Dodatkowych pakietów, oficjalnie dostępnych w repozytorium pakietów CRAN, jest na dziś dzień ponad 6000. Są to pakiety specjalistyczne, dla bioinformatyków, botaników, historyków, lingwistów, osób zainteresowanych obliczeniami rozproszonymi i dla wielu innych specjalistycznych zastosowań.

My, na potrzeby tego kursu, będziemy korzystać z pakietu `PogromcyDanych`, w którym umieszczone są dodatkowe

zbiory danych, na których będziemy pracować. Instalacja tego pakietu automatycznie wywoła również instalację wszystkich pakietów zależnych, dzięki czemu jedną linijką możemy zainstalować wszystko, co nam będzie potrzebne w tym kursie.

Instalację nowego pakietu wykonuje się funkcją `install.packages("nazwa_pakietu")`, którą należy uruchomić w linii poleceń programu R. Tak więc po instalacji R i RStudio, powinniśmy uruchomić program RStudio. Następnie w konsoli (oknie z nazwą `Console`) wpisać polecenie i zakończyć je klawiszem ENTER.

```
install.packages("PogromcyDanych")  
library(PogromcyDanych)
```

Pobierze i zainstaluje ono z Internetu wszelkie niezbędne pakiety i zbiory danych z których będziemy korzystać w tym kursie.

*Uwaga* O ile polecenie `install.packages()` instaluje na komputerze odpowiedni pakiet, to przed jego użyciem za każdym razem powinniśmy pakiet włączyć poleceniem `library()`.

```
> install.packages("SmarterPoland")
trying URL 'http://cran.rstudio.com/bin/macosx/mavericks/contrib/3.1/SmarterPoland_1.2.tgz'
Content type 'application/x-gzip' length 24086 bytes (23 Kb)
opened URL
=====
downloaded 23 Kb
```

```
The downloaded binary packages are in
/var/folders/_l/jllqh32s3llbxmtrh2431vpr0000gn/T//Rtmpv0SSXs/downloaded_packages
> |
```

---

## Gotowi do pracy

Zainstalowaliśmy program R, RStudio i pakiet  
PogromcyDanych?

Jeżeli tak to jesteśmy gotowi do rozpoczęcia pracy z R!

Jeżeli chcemy sprawdzić czy wszystko poprawnie się  
zainstalowało, to po uruchomieniu programu RStudio w  
okienku o tytule `Console` należy wpisać

```
demo(graphics)
```

i nacisnąć ENTER. Funkcja `demo()` wyświetla pokazowe  
wykresy programu R. Aby przejść do kolejnego należy w  
konsoli nacisnąć przycisk ENTER. Aby przerwać  
działanie funkcji `demo()` należy nacisnąć przycisk ESC.

# Wprowadzenie do R i RStudio

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 4*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Jak pracować z RStudio](#)
- [Interaktywna praca z konsolą](#)
- [Zmienne i ich wartości w pamięci programu RStudio](#)
- [Funkcje](#)
- [Jak wykonać sekwencję liczb?](#)
- [Wykres z morałem](#)
- [Zadanie](#)

Ten odcinek jest też dostępny w formacie video na kanale youtube <https://youtu.be/QBAVcoXAS98>

## O czym jest ten odcinek

Najtrudniejsze są początki. Aby dojść do miejsca, w

którym nasza praca stanie się przyjemnością i będzie efektywna, musimy nauczyć się nowego języka. A nauka, czy to języka, czy czegokolwiek innego, to seria prób, czasem niepowodzeń, kolejnych prób i tak nieustannie.

Prawdziwa nauka rozpocznie się z odcinkiem piątym, a więc kolejnym. Ale zanim będziemy gotowi rozpocząć naukę musimy oswoić się z narzędziem, z którym przyjdzie nam pracować.

W tym odcinku nauczymy się:

- w jaki sposób wprowadzać instrukcje do programu R,
- w jaki sposób definiować nowe zmienne w programie R,
- jak szukać pomocy dla funkcji.

Pracując ze slajdami, możemy w każdej chwili nacisnąć przycisk **T**, rozwinię on spis treści, dzięki czemu łatwo nam będzie przejść do pożądanego slajdu. Możemy również w każdej chwili nacisnąć przycisk **A**, aby zamienić slajdy na ciągły dokument.

---

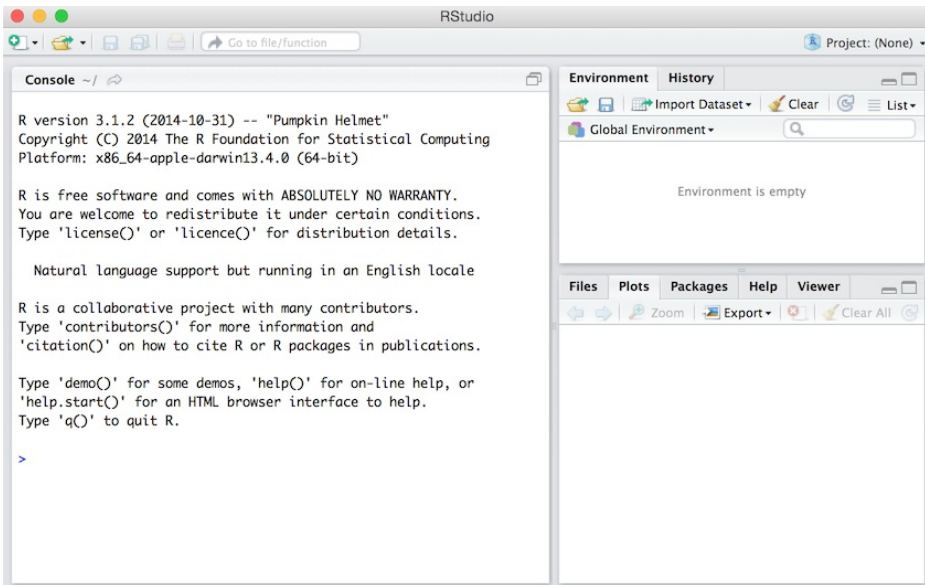
## Jak pracować z RStudio

Zacznijmy naszą przygodę od krótkiego omówienia programu RStudio. Będziemy z nim pracować przez większość czasu, warto go bliżej poznać.

Po zainstalowaniu przy pierwszym otwarciu program RStudio będzie wyglądał w poniżej przedstawiony sposób.

Po lewej stronie jest interaktywna konsola, po prawej stronie są panele pomocnicze, które omówimy za kilka minut.

Konsolę otwiera zbiór informacji o wersji zainstalowanego programu R, oraz znak zachęty `>`. Ten znak, oznacza, że program R jest gotowy do dalszej pracy i możemy wpisywać mu polecenia.



## Interaktywna praca z konsolą

Zazwyczaj, praca z programem R przebiega w sposób interaktywny.

Jeżeli wpisujemy komendę do konsoli i zatwierdzimy ją klawiszem Enter, to R wykona polecenie i jego wynik wyświetli w konsoli.

Przećwiczmy to na kilku prostych operacjach arytmetycznych.

```
2 + 2
```

```
## [1] 4
```

```
2^10
```

```
## [1] 1024
```

Na slajdach wyniki instrukcji wprowadzanych do programu R przedstawimy poprzedziwszy je znakami `##`. Poniżej znajduje się obraz pokazujący jak wygląda konsola.

```
> 2 + 2
[1] 4
> 2^10
[1] 1024
>
> |
```

## Zmienne i ich wartości w pamięci programu RStudio

Wyniki operacji możemy przypisać do zmiennych. Dzięki temu, zamiast wyświetlać się na ekranie, zostaną one zapamiętane w pamięci programu R, oraz będziemy mogli je wykorzystywać w przyszłości.

Wartość do zmiennej przypisuje się zazwyczaj z użyciem jednego z operatorów `=`, `<-` lub `->`. Możemy je stosować

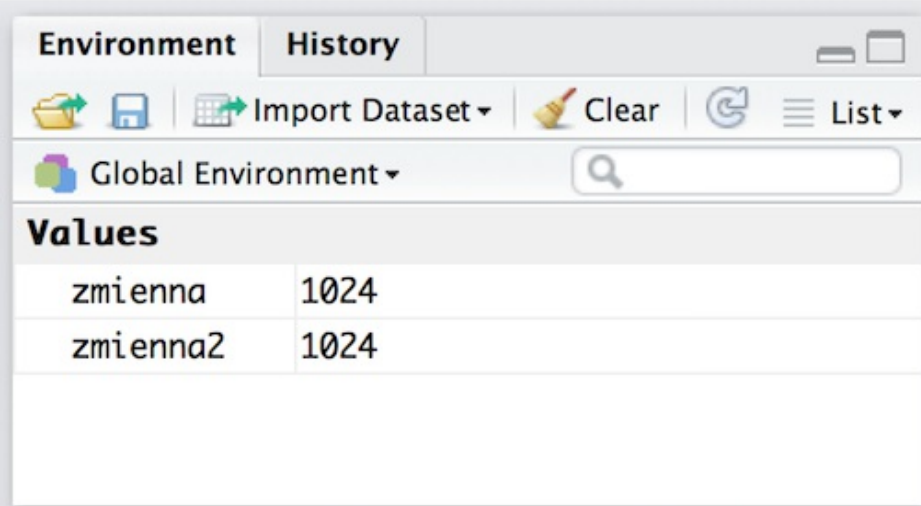


zamiennie, ale stosowanie strzałki `<-` pozwoli nam zachować większą czytelność w przyszłości przy bardziej złożonych poleceniach.

```
zmienna = 2^10  
zmienna2 <- 2^10
```

Zauważmy, że w oknie po prawej górnej stronie wyświetliły się nowo zdefiniowane zmienne oraz ich wartości.

Okno to pozwala w każdej chwili sprawdzić, jakie zmienne program R aktualnie pamięta, oraz jakie są wartości tych zmiennych.



The screenshot shows the R Environment window with the 'Environment' tab selected. The 'Global Environment' is expanded, showing a list of variables. The variables 'zmienna' and 'zmienna2' are listed with their respective values, both of which are 1024.

Values	
zmienna	1024
zmienna2	1024

## Funkcje

Prawdziwa siła programu R tkwi w dużym zbiorze funkcji, którymi możemy wykonywać najróżniejsze operacje. Funkcjami możemy analizować dane, rysować dane, odsłuchiwać dane, wykonywać obliczenia, wysyłać maile, robić najróżniejsze rzeczy.

Oczywiście musimy znać nazwy funkcji, które to wszystko robią.

Przypuśćmy, że chcemy zbudować sekwencje liczb od 0 do 10 z krokiem co 0,1. Jak się wkrótce okaże, takie sekwencje są bardzo przydatne.

Operatorem `??` możemy wyszukać funkcję, która w opisie ma określone słowo lub zwrot.

```
?? "sequence"
```

Okazuje się, że jest kilka funkcji, które mają związek z sekwencjami. Ta która nas interesuje to `seq()`. Znajduje się ona w pakiecie `base`. Wszystkie funkcje są pogrupowane w pakiety, a pakiet `base` zawiera funkcje do podstawowych operacji.

Aby wyświetlić opis dla zadanej funkcji wystarczy wykorzystać operator `?`.

```
?seq
```

Opis funkcji składa się z opisu jej argumentów, szczegółowego opisu działania funkcji oraz przykładów jej użycia. Te przykłady można skopiować i wkleić do konsoli, aby zobaczyć co jest ich wynikiem.

## Jak wykonać sekwencję liczb?

Dla funkcji `seq()`, aby podać początek, koniec i krok dla sekwencji, należy wskazać argumenty `from`, `to` i `by`. Możemy więc stworzyć sekwencję za pomocą następującej instrukcji.

```
sekwencja <- seq(from = 0, to = 10, by = 0.1)
sekwencja
```

##	[1]	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0
##	[15]	1.4	1.5	1.6	1.7	1.8	1.9	2.0	2
##	[29]	2.8	2.9	3.0	3.1	3.2	3.3	3.4	3
##	[43]	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4
##	[57]	5.6	5.7	5.8	5.9	6.0	6.1	6.2	6
##	[71]	7.0	7.1	7.2	7.3	7.4	7.5	7.6	7
##	[85]	8.4	8.5	8.6	8.7	8.8	8.9	9.0	9
##	[99]	9.8	9.9	10.0					

Zauważmy, że wartości do argumentów przypisujemy znakiem `=`. To przypisanie ma jednak inne znaczenie. Nie tworzymy nowej zmiennej o nazwie `from` ale wskazujemy, że argument funkcji `'from'` ma mieć wartość 0. Dla większej czytelności, do przypisania, które tworzy zmienną w pamięci R, będziemy używać operatora `<-`, a

do przypisania wartości do argumentu funkcji będziemy używać operatora `=`.

Argumenty do funkcji możemy podać w dowolnej kolejności. Możemy najpierw określić argument `by` a następnie `from` i `to`.

```
sekwencja <- seq(by = 0.1, from = 0, to = 10)
```

Jeżeli podajemy argumenty w domyślnej kolejności, opisanej w pliku pomocy, wtedy możemy pominąć podawanie ich nazw. Program R z kolejności argumentów odczyta, który co opisuje.

```
sekwencja <- seq(0, 10, 0.1)
```

Mamy już sekwencję liczb. Zróbmy z nią coś ciekawego.

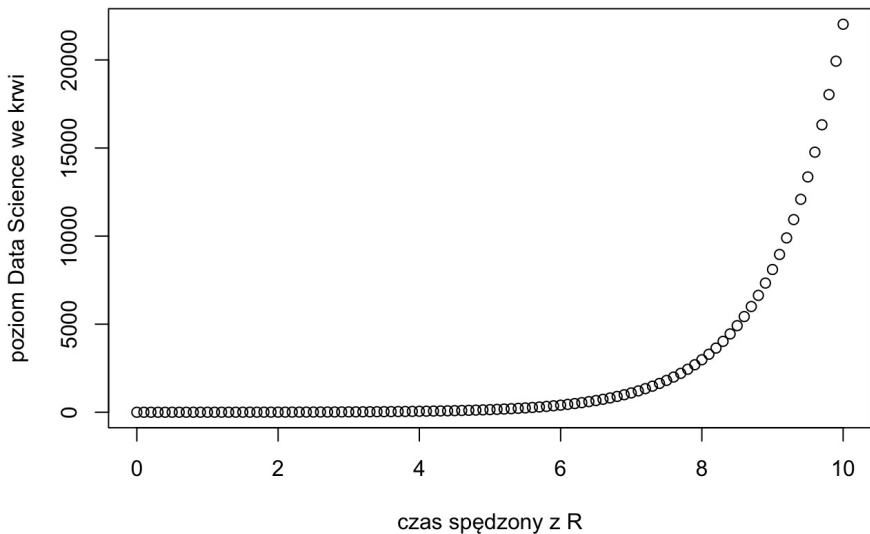
## Wykres z morałem

Funkcją `exp()` policzmy eksponentę (*eksponenta, inaczej funkcja wykładnicza, to bardzo szybko rosnąca funkcja matematyczna*) z sekwencji, a następnie - wykorzystując funkcję `plot()` - narysujmy i sekwencję, i jej eksponentę. Zauważmy, że argumentem tej funkcji jest wektor, a ponieważ większość operacji w R jest wektoryzowana, funkcja `exp()` policzy eksponentę dla każdego elementu wektora `sekwencja`.

Poniższy wykres, to bardzo ważny wykres, ponieważ przedstawia bardzo ważną informację dotyczącą uczenia się. Pierwsze godziny poznawania R poświęcimy na zbudowanie bazy funkcji R. Będzie to okres pierwszych frustracji, gdy efekty będą relatywnie małe, a wysiłek z naszej strony będzie duży. Ale po kilku pierwszych godzinach, jeżeli tylko wytrwamy, okaże się, że nasze możliwości rosną wykładniczo.

```
poziom <- exp(sekwencja)

plot(x = sekwencja, y = poziom,
      xlab="czas spędzony z R", ylab="poziom Data Science we krwi")
```



# Zadanie

Pod koniec większości odcinków przygotowaliśmy kilka zadań do wykonania.

Pierwsze zadania są proste, kolejne coraz trudniejsze. Warto zmierzyć się z nimi, gdyż najlepszym sposobem by nauczyć się języka programowania jest ćwiczenie, ćwiczenie i jeszcze raz ćwiczenie.

Odpowiedzi do zadań umieszczone są na stronie kursu, można je w każdej chwili sprawdzić.

Zadanie do tego odcinka polega na uruchomieniu wszystkich przedstawionych instrukcji w programie RStudio i sprawdzeniu czy otrzymuje się te same wyniki.

# Jak wczytać dane do programu R

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 5*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Dane tabelaryczne w plikach tekstowych](#)
- [Dane tabelaryczne w plikach tekstowych](#)
- [Wczytywanie danych do R](#)
- [Dane tabelaryczne w plikach tekstowych](#)
- [Dane tabelaryczne w plikach tekstowych](#)
- [Dane tabelaryczne w plikach tekstowych](#)
- [Dane tabelaryczne w plikach tekstowych](#)
- [Dane w formacie programu Excel](#)
- [Dane w formacie programu Excel](#)
- [Dane w formacie programu Excel](#)
- [Dane binarne programu R](#)
- [Wczytywanie danych z pakietów R](#)
- [Wczytywanie danych z pakietów R](#)
- [Wczytywanie danych poprzez web API](#)

- [Podsumowanie instrukcji R](#)
- [Zadanie](#)

## O czym jest ten odcinek

Zanim rozpoczniemy analizę danych, musimy najpierw wczytać dane do programu R.

Dane mogą być przechowywane w najróżniejszych formatach, takich jak baza danych, plik tekstowy, plik w formacie programu Excel (nowszy .xlsx lub starszy .xls) lub plik w formacie innego programu do analizy danych (SAS, SPSS, itp).

W tym odcinku odpowiemy na pytania:

- Jak wczytywać dane tabelaryczne z plików tekstowych?
  - Jak wczytywać dane tabelaryczne z plików Excela [xls lub xlsx]?
  - Jak wczytywać dane z pakietów R?
  - Gdzie szukać informacji o tym jak wczytać dane z innych formatów / programów?
- 

## Dane tabelaryczne w plikach



# tekstowych

Jednym z częstszych formatów używanych do przechowywania i wymiany danych są pliki tekstowe.

Nazwa ‘plik tekstowy’ bierze się stąd, że treść tego pliku możemy otworzyć w standardowym edytorze takim jak Notepad/Notatnik w Windowsie lub vim w Linuxie/OSX. Pliki tekstowe można również otwierać z użyciem programu RStudio (uwaga, duże pliki mogą się długo otwierać). Jeżeli często pracujemy z dużymi lub wieloma plikami tekstowymi to warto wyposażyć się w dobre narzędzie do pracy z nimi, np. edytor [Sublime Text](#).

Przyjrzymy się tabeli z danymi zapisanymi w pliku [http://biecek.pl/MOOC/dane/koty\\_ptaki.csv](http://biecek.pl/MOOC/dane/koty_ptaki.csv)

Zawartość tego pliku wygląda następująco:

```
gatunek;waga;dlugosc;predkosc;habitat;zywotnos  
Tygrys;300;2,5;60;Azja;25;Kot  
Lew;200;2;80;Afryka;29;Kot  
Jaguar;100;1,7;90;Ameryka;15;Kot  
Puma;80;1,7;70;Ameryka;13;Kot  
Leopard;70;1,4;85;Azja;21;Kot  
Gepard;60;1,4;115;Afryka;12;Kot  
Irbis;50;1,3;65;Azja;18;Kot  
Jerzyk;0,05;0,2;170;Euroazja;20;Ptak  
Strus;150;2,5;70;Afryka;45;Ptak  
Orzel przedni;5;0,9;160;Polnoc;20;Ptak
```

Sokol wedrowny;0,7;0,5;110;Polnoc;15;Ptak  
Sokol norweski;2;0,7;100;Polnoc;20;Ptak  
Albatros;4;0,8;120;Poludnie;50;Ptak

# Dane tabelaryczne w plikach tekstowych

W jaki sposób dane tabelaryczne zapisane są w tym pliku?

Wyrównując wartości, zauważymy, że logiczna struktura tego pliku to tabela wartości rozdzielonych średnikami.

gatunek	waga	dlugosc	predkosc	habitat	zywotnosc	druzyna
Tygrys	300	2,5	60	Azja	25	Kot
Lew	200	2	80	Afryka	29	Kot
Jaguar	100	1,7	90	Ameryka	15	Kot
Puma	80	1,7	70	Ameryka	13	Kot
Leopard	70	1,4	85	Azja	21	Kot
Gepard	60	1,4	115	Afryka	12	Kot
Irbis	50	1,3	65	Azja	18	Kot
Jerzyk	0,05	0,2	170	Euroazja	20	Ptak
Strus	150	2,5	70	Afryka	45	Ptak
Orzel przedni	5	0,9	160	Polnoc	20	Ptak
Sokol wedrowny	0,7	0,5	110	Polnoc	15	Ptak
Sokol norweski	2	0,7	100	Polnoc	20	Ptak
Albatros	4	0,8	120	Poludnie	50	Ptak

Zauważmy, że:

- pierwszy wiersz to nagłówek - zawiera nazwy kolumn rozdzielane średnikiem  
(gatunek;waga;dlugosc;predkosc;habitat;zywotnosc;druzyna)

- kolejne wiersze przedstawiają dane w postaci wektora wartości, które są rozdzielane znakiem ; (średnik),
  - w pliku występują liczby, które nie są liczbami całkowitymi - separatorem dziesiętnym jest znak , (przecinek).
- 

## Wczytywanie danych do R

Instrukcja wczytująca dane do R składa się zazwyczaj z trzech członów.

Ostatnim (po prawej stronie na poniższym schemacie) jest funkcja, która odczytuje dane, przetwarza i zamienia na postać zrozumiałą dla programu R (na poniższym schemacie ta funkcja to `read.table()`).

Aby móc na tych danych operować, należy nadać im jakąś nazwę (lewy człon na poniższym schemacie). Tę nazwę określa się zazwyczaj terminem *zmienna*.

Operację przypisania wartości odczytanych przez funkcję do zmiennej można wykonać operatorem `<-`, `=` lub `->` (patrz poprzedni odcinek).

W programie R wynik funkcji, o ile nie jest przypisany do

zmiennej, jest wyświetlany na ekranie. Jeżeli chcemy wynik działania funkcji wykorzystać w przyszłości, to należy go do zmiennej przypisać.

Nazwa zmiennej

Operator przypisania <- lub =

Funkcja wczytująca dane

koty\_ptaki

<-

read.table("koty\_ptaki.csv", header=TRUE, sep=";")

gatunek	waga	dlugosc	predkosc	habitat	zywnosc	druzyna
Tygris	300	2.5	60	Azja	25	Kot
Lew	200	2	80	Afryka	29	Kot
Jaguar	100	1.7	90	Ameryka	15	Kot
Puma	80	1.7	70	Ameryka	13	Kot
Leopard	70	1.4	85	Azja	21	Kot
Gepard	60	1.4	115	Afryka	12	Kot
Irbis	50	1.3	65	Azja	18	Kot
Jerzyk	0.05	0.2	170	Europa	20	Ptak
Strus	150	2.5	70	Afryka	45	Ptak
Orzeł przedni	5	0.9	160	Połnoc	20	Ptak
Sokol wędrowny	0.7	0.5	110	Połnoc	15	Ptak
Sokol norweski	2	0.7	100	Połnoc	20	Ptak
Albatros	4	0.8	120	Południe	50	Ptak

Więcej informacji o tym jak przypisywać wartości do zmiennych przedstawionych jest w odcinku 4 *Interaktywna praca z R*.

# Dane tabelaryczne w plikach tekstowych

Zajmijmy się na razie prawą stroną przedstawionego schematu, a więc funkcją wczytującą dane.

Funkcja `read.table()` ma wiele argumentów (patrz kolejny slajd), ale nie musimy ich wszystkich określać. Wystarczy wskazać nazwę pliku oraz te argumenty, które powinny mieć inne wartości niż domyślne.

W przypadku rozważanego pliku musimy określić argumenty

`file = "http://biecek.pl/MOOC/dane/koty_ptaki..."` (ścieżka do pliku tekstowego, w tym przypadku czytamy dane bezpośrednio z internetowego adresu), `sep=";"` (separator kolejnych pól będzie średnik), `dec=","` (separatorem dziesiętnym jest przecinek), `header=TRUE` (pierwszy wiersz ma nagłówek). Ponieważ wynik tej funkcji nie jest do niczego przypisany, dlatego wczytany zbiór danych jest wyświetlany na ekranie.

```
read.table(file = "http://biecek.pl/MOOC/dane/koty_ptaki.txt",
           sep=";", dec=",", header=TRUE)
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	z
## 4		Puma	80.00	1.7	70	z
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Et
## 9		Strus	150.00	2.5	70	
## 10	Orzel	przedni	5.00	0.9	160	
## 11	Sokol	wedrowny	0.70	0.5	110	
## 12	Sokol	norweski	2.00	0.7	100	
## 13		Albatros	4.00	0.8	120	Po

# Dane tabelaryczne w plikach

# tekstowych

Do wczytywania danych z pliku tekstowego posłużyć może funkcja `read.table()`. Ponieważ pliki tekstowe mogą mieć bardzo różną postać, funkcja ta ma wiele argumentów. Wybrane argumenty przedstawia poniższa deklaracja (jest ich więcej, ale część pominiemy by nie zaciemniać obrazu).

```
read.table(file, header = FALSE, sep = "", dec
           skip = 0, comment.char = "#",
           stringsAsFactors = default.stringsAs
```

Ponieważ dane mogą być zapisane z użyciem różnych formatów, argumenty tej funkcji określają format danych do wczytania. Znaczenie kolejnych argumentów podane jest poniżej.

- `file` - ścieżka do pliku z danymi, może być też adres URL. Jeżeli wartości argumentów podajemy w ich domyślnej kolejności, to możemy pominąć nazwy argumentów (dlatego w kolejnych przykładach nie będziemy podawać nazwy tego argumentu).
- `header` - flaga określająca, czy pierwszy wiersz należy traktować jako nagłówek. W naszym przypadku `header=TRUE`.
- `sep` - znak rozdzielający kolumny. W naszym przypadku `sep=";"`, ale popularnymi separatorami

są również znak tabulacji (oznaczany `\t`), przecinek lub spacja. Separatorem może być dowolny, ale tylko jeden znak.

- `dec` - separator dziesiętny. Zazwyczaj jest to `.` lub `,`. W naszym przypadku `dec=" , "`.
  - `nrows` - maksymalna liczba wierszy do wczytania. Domyślnie ten argument przyjmuje wartość `-1`, czyli wczytaj wszystkie wiersze.
  - `skip` - liczba pierwszych wierszy do pominięcia przy wczytywaniu danych, domyślnie `0`, czyli nie pomijaj żadnego wiersza.
  - `comment.char` - znak komentarza. Jeżeli w danych wystąpi ten znak to treść od tego znaku do końca linii będzie zignorowana.
  - `stringsAsFactors` - czy napisy powinny być domyślnie przekształcone w zmienne jakościowe. Domyślnie `TRUE`, o konsekwencjach tego przekształcenia napiszemy w odcinku 11 *Cechy jakościowe*.
- 

## Dane tabelaryczne w plikach tekstowych

Wynikiem funkcji `read.table()` jest tabelaryczny zbiór danych, który w R nazywa się *ramką danych* (ang.

data.frame).

Aby na nim pracować, musimy wynik funkcji `read.table()` zapamiętać w zmiennej. Na przykład w zmiennej o nazwie `koty_ptaki`.

```
koty_ptaki <- read.table("http://biecek.pl/MOOC/
                        sep=";", dec=",", head=
## po wpisaniu nazwy zmiennej, jako wynik wyświetli
koty_ptaki
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	z
## 4		Puma	80.00	1.7	70	z
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	En
## 9		Strus	150.00	2.5	70	
## 10	Orzel	przedni	5.00	0.9	160	
## 11	Sokol	wedrowny	0.70	0.5	110	
## 12	Sokol	norweski	2.00	0.7	100	
## 13		Albatros	4.00	0.8	120	Pe

## Dane tabelaryczne w plikach tekstowych

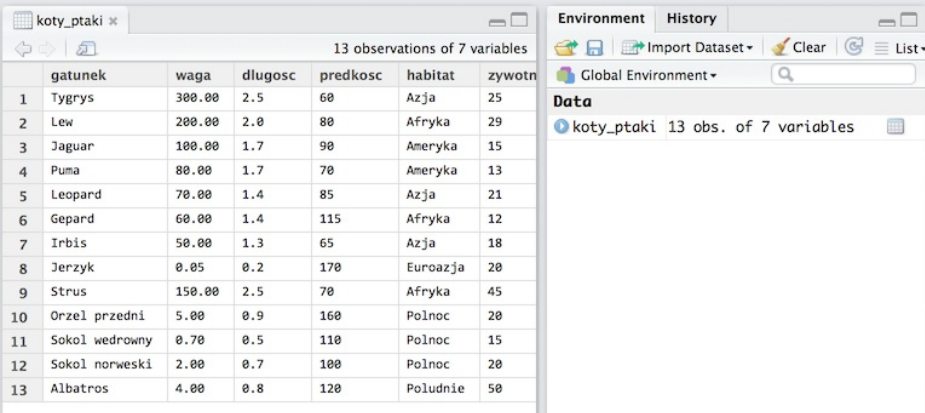
W programie R Studio w prawym górnym oknie,



zatytułowanym *Environment* wyświetlane są wszystkie zdefiniowane zmienne. Po poprawnym wczytaniu danych, powinniśmy je widzieć właśnie w tym oknie.

Dwukrotne kliknięcie na wskazaną zmienną powoduje otwarcie okna prezentującego zawartość zmiennej. W ten sposób możemy szybko podejrzeć co wczytało się do zmiennej `koty_ptaki`.

Z przyczyn wydajnościowych wyświetlanych jest tylko pierwsze 1000 wierszy i kilkaset kolumn. Tak więc dla dużych zbiorów danych wyświetlony będzie tylko fragment całego zbioru.



The screenshot shows the RStudio interface. The top-left pane is titled 'koty\_ptaki' and displays '13 observations of 7 variables'. The top-right pane is titled 'Environment' and shows 'Global Environment' with a search bar. Below the search bar, the 'Data' section lists 'koty\_ptaki' with '13 obs. of 7 variables'. The bottom pane shows a table with 13 rows and 7 columns.

	gatunek	waga	dlugosc	predkosc	habitat	zywotn
1	Tygrys	300.00	2.5	60	Azja	25
2	Lew	200.00	2.0	80	Afryka	29
3	Jaguar	100.00	1.7	90	Ameryka	15
4	Puma	80.00	1.7	70	Ameryka	13
5	Leopard	70.00	1.4	85	Azja	21
6	Gepard	60.00	1.4	115	Afryka	12
7	Irbis	50.00	1.3	65	Azja	18
8	Jerzyk	0.05	0.2	170	Euroazja	20
9	Strus	150.00	2.5	70	Afryka	45
10	Orzeł przedni	5.00	0.9	160	Polnoc	20
11	Sokol wedrowny	0.70	0.5	110	Polnoc	15
12	Sokol norweski	2.00	0.7	100	Polnoc	20
13	Albatros	4.00	0.8	120	Poludnie	50

## Dane w formacie programu Excel

Innym popularnym formatem przechowywania danych są

pliki w formacie Excela [rozszerzenia xls lub.xlsx].

W programie R dostępnych jest kilka pakietów pozwalających na odczytywanie danych w formacie Excela. Najpopularniejsze pakiety to `gdata`, `xlsReadWrite`, `XLConnect`, `xlsx`. Różnią się one zewnętrznymi bibliotekami, które wykorzystują, przez co z niektórych może być łatwiej skorzystać pod Windowsem z innych pod Linuxem.

W tym odcinku wykorzystamy funkcję `read.xls()` z pakietu `gdata`. Wymaga ona zainstalowanego programu `perl` (jeżeli nie jest on jeszcze zainstalowany, należy go doinstalować). Pakiet `gdata` nie jest instalowany z podstawową dystrybucją R, dlatego przed pierwszym użyciem należy go zainstalować poleceniem `install.packages("gdata")`.

Dane, które chcemy wczytać, w Excelu są dostępne pod adresem [http://biecek.pl/MOOC/dane/koty\\_ptaki.xls](http://biecek.pl/MOOC/dane/koty_ptaki.xls) i wyglądają następująco.

	A	B	C	D	E	F	G
1	gatunek	waga	dlugosc	predkosc	habitat	zywnosc	druzyna
2	Tygrys	300	2,5	60	Azja	25	Kot
3	Lew	200	2	80	Afryka	29	Kot
4	Jaguar	100	1,7	90	Ameryka	15	Kot
5	Puma	80	1,7	70	Ameryka	13	Kot
6	Leopard	70	1,4	85	Azja	21	Kot
7	Gepard	60	1,4	115	Afryka	12	Kot
8	Irbis	50	1,3	65	Azja	18	Kot
9	Jerzyk	0,05	0,2	170	Euroazja	20	Ptak
10	Strus	150	2,5	70	Afryka	45	Ptak
11	Orzel przedni	5	0,9	160	Polnoc	20	Ptak
12	Sokol wedrowny	0,7	0,5	110	Polnoc	15	Ptak
13	Sokol norweski	2	0,7	100	Polnoc	20	Ptak
14	Albatros	4	0,8	120	Poludnie	50	Ptak

# Dane w formacie programu Excel

Funkcja `read.xls()` oczekuje dwóch argumentów: ścieżki do pliku i argumentu `sheet`, którym wskazuje, którą zakładkę z pliku Excela należy odczytać.

```
library(gdata)
read.xls("http://biecek.pl/MOOC/dane/koty_ptak:
```

##	gatunek	waga	dlugosc	predkosc	1
## 1	Tygrys	300.00	2.5	60	
## 2	Lew	200.00	2.0	80	
## 3	Jaguar	100.00	1.7	90	2
## 4	Puma	80.00	1.7	70	2
## 5	Leopard	70.00	1.4	85	
## 6	Gepard	60.00	1.4	115	
## 7	Irbis	50.00	1.3	65	

##	8	Jerzyk	0.05	0.2	170	Et
##	9	Strus	150.00	2.5	70	
##	10	Orzel przedni	5.00	0.9	160	
##	11	Sokol wedrowny	0.70	0.5	110	
##	12	Sokol norweski	2.00	0.7	100	
##	13	Albatros	4.00	0.8	120	Pc

	A	B	C	D	E	F	G
1	gatunek	waga	dlugosc	predkosc	habitat	zywnosc	druzyna
2	Tygrys	300	2,5	60	Azja	25	Kot
3	Lew	200	2	80	Afryka	29	Kot
4	Jaguar	100	1,7	90	Ameryka	15	Kot
5	Puma	80	1,7	70	Ameryka	13	Kot
6	Leopard	70	1,4	85	Azja	21	Kot
7	Gepard	60	1,4	115	Afryka	12	Kot
8	Irbis	50	1,3	65	Azja	18	Kot
9	Jerzyk	0,05	0,2	170	Euroazja	20	Ptak
10	Strus	150	2,5	70	Afryka	45	Ptak
11	Orzel przedni	5	0,9	160	Polnoc	20	Ptak
12	Sokol wedrowny	0,7	0,5	110	Polnoc	15	Ptak
13	Sokol norweski	2	0,7	100	Polnoc	20	Ptak
14	Albatros	4	0,8	120	Poludnie	50	Ptak

## Dane w formacie programu Excel

Dlaczego dostępnych jest tak wiele pakietów do wczytywania danych z Excela do R? Otóż, żadne z rozwiązań nie jest wyraźnie lepsze od pozostałych. Jedne są szybsze, inne lepiej radzą sobie z dużymi danymi.

Zestawienie silnych i słabych stron poszczególnych pakietów znajduje się na stronie

<http://www.thertrader.com/2014/02/11/a-million-ways-to->

[connect-r-and-excel/](#). Wybrane fragmenty porównania wymieniamy poniżej.

- `XLConnect` ma dużo możliwości, szczególnie dotyczących konwersji typów, ale jest wolniejszy niż inne rozwiązania i wymagający jeżeli chodzi o RAM, przez co może nie poradzić sobie z dużymi zbiorami danych.
- `gdata` ma kilka użytecznych funkcji, pozwalających na nawigację po skoroszytach, np. `sheetCount()` i `sheetNames()`

Wczytując dane z plików, które zostały stworzone na innym systemie operacyjnym lub w innej lokalizacji, trzeba liczyć się z różnicami dotyczącymi kodowania polskich znaków lub np. kropek dziesiętnych w liczbach rzeczywistych. Jeżeli ten problem może nas dotyczyć, to warto zapoznać się z kartą „Dobrych Praktyk” dostępnych na stronie <http://withr.me/configure-character-encoding-for-r-under-linux-and-windows/>

Zestawienie najpopularniejszych pakietów wczytujących dane z Excela prezentuje poniższa tabela.

<b>pakiet</b>	<b>funkcja_odczyt</b>	<b>funkcja_zapis</b>	<b>formaty</b>	<b>fu</b>
xlsx	read.xlsx()	write.xlsx()	.xlsx, .xls	śr

openxlsx	read.xlsx()	write.xlsx()	.xlsx	śr
gdata	read.xls()	–	.xlsx, .xls	m
WriteXLS	–	WriteXLS()	.xlsx, .xls	m
XLConnect	readWorksheet()	writeWorksheet()	.xlsx, .xls	du

---

## Dane binarne programu R

Natywnym formatem dla programu R są pliki binarne o rozszerzeniu `rda` lub `RData`.

Dane w tym formacie są skompresowane, przez co zajmują mniej miejsca na dysku niż w pliku tekstowym lub formacie Excela. Wadą jest to, że niewiele programów poza programem R potrafi je odczytać.

Dane w tym formacie można wczytać do programu R poleceniem `load()`. Pierwszym argumentem tej funkcji jest ścieżka do pliku z danymi. Jeżeli chcemy dane odczytać z internetu, należy dodatkowo użyć funkcji `url()`.

*Uwaga!* W przeciwieństwie do wcześniej poznanych funkcji, funkcja `load()` nie zwraca zbioru danych jako

wynik, ale ładuje zbiór danych bezpośrednio do przestrzeni nazw (dane zapisane są wraz z nazwą zmiennej). Dlatego w poniższym przykładzie nie ma instrukcji przypisania do zmiennej `koty_ptaki`. Nazwa tej zmiennej jest pamiętana wewnątrz pliku `rda`.

```
load(url("http://biecek.pl/MOOC/dane/koty_ptaki:
koty_ptaki
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	z
## 4		Puma	80.00	1.7	70	z
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	
## 10	Orzel	przedni	5.00	0.9	160	
## 11	Sokol	wedrowny	0.70	0.5	110	
## 12	Sokol	norweski	2.00	0.7	100	
## 13		Albatros	4.00	0.8	120	Pe

# Wczytywanie danych z pakietów R

Przygotowując ten kurs musiałem zmierzyć się z następującym problemem. Jak w najprostszy sposób udostępnić uczestnikom kursu kilka zbiorów danych? Rozważając różne opcje, stwierdziłem że najłatwiejsza to

udostępnić dane z użyciem pakietu dla programu R.

Pakiety to zbiory funkcji oraz zbiorów danych. Można takie pakiety wygodnie przygotowywać oraz udostępniać innym osobom. Istnieją pakiety zawierające tylko funkcje, wyłącznie zbiory danych lub oba te zestawy.

Aby sprawdzić jakie zbiory danych są dostępne w określonym pakiecie, można wykorzystać funkcję `data()` z argumentem `package`. Zobaczmy jakie zbiory danych udostępnione są w pakiecie `PogromcyDanych`.

```
data(package="PogromcyDanych")
```

**Uwaga!** Pakiet `PogromcyDanych` nie jest dostępny w podstawowej dystrybucji programu R i trzeba go wcześniej zainstalować poleceniem

`install.packages()`. Szczegółowa instrukcja, jak to zrobić jest przedstawiona w odcinku 3 *Jak zainstalować R, RStudio oraz dodatkowe pakiety?*.

Data sets in package 'SmarterPoland':

BDLtree	
auta2012	Over 210 thousands offers of used cars from otoMoto website
galton	Galton's and Pearson's height data for parents and children in centimeters
koty	Size, habitat, speed and weight of big cats
mandatySejmik2014	
pearson	Galton's and Pearson's height data for parents and children in centimeters
skiJumps2013	Results from ski jumps, season 2013/2014
skiJumps2013labels	Results from ski jumps, season 2013/2014



# Wczytywanie danych z pakietów R

Jednym ze zbiorów danych udostępnionych w tym pakiecie jest zbiór danych `koty_ptaki`.

Po wczytaniu pakietu, ten zbiór danych jest dostępny bez potrzeby stosowania dodatkowych instrukcji. Można go wyświetlić na ekranie wpisując do konsoli jego nazwę.

```
library(PogromcyDanych)
koty_ptaki
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	z
## 4		Puma	80.00	1.7	70	z
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Et
## 9		Strus	150.00	2.5	70	
## 10	Orzel przedni		5.00	0.9	160	
## 11	Sokol wedrowny		0.70	0.5	110	
## 12	Sokol norweski		2.00	0.7	100	
## 13	Albatros		4.00	0.8	120	Po

---

## Wczytywanie danych poprzez web API

Pewne dane dostępne są przez serwisy z danymi poprzez

API (ang. Application Programming Interface). Oznacza to, że nie możemy pobrać wszystkich danych z bazy danych, ale możemy tą bazę odpytywać o określone informacje.

W ten sposób można dostać się do danych takich gigantów jak WorldBank, Eurostat czy osatnio GUS.

Przykładem niech będzie baza Eurostat, do której można sięgać używając funkcji `getEurostatRCV()` z pakietu `SmarterPoland`. W bazie Eurostat tabela `tsdtr210` zawiera dane o ilości pasażero-kilometrów dla różnych krajów i środków transportu. Ta tabela jest dostępna w pliku <http://ec.europa.eu/eurostat/tgm/table.do?tab=table&init=1&language=en&pcode=tsdtr210&plugin=>

Używając API udostępnionego przez pakiet `SmarterPoland` pokażemy jak pobierać te dane bezpośrednio do R.

*Uwaga!* Pakiet `SmarterPoland` nie jest dostępny w podstawowej dystrybucji programu R i trzeba go wcześniej zainstalować poleceniem `install.packages()`. Szczegółowa instrukcja, jak to zrobić jest przedstawiona w odcinku 3 *Jak zainstalować R, RStudio oraz dodatkowe pakiety?*.

```
library(SmarterPoland)
tsdtr210 <- getEurostatRCV("tsdtr210")
```

```
head(tsdtr210, 3)

##                                unit vehicle geo time va
## PC_BUS_TOT_AT_1990          PC BUS_TOT  AT 1990
## PC_BUS_TOT_BE_1990          PC BUS_TOT  BE 1990
## PC_BUS_TOT_BG_1990          PC BUS_TOT  BG 1990
```

```
summary(tsdtr210)

##      unit              vehicle              geo
## PC:2415  BUS_TOT:805    AT      : 69 1990
##          CAR      :805    BE      : 69 1990
##          TRN      :805    BG      : 69 1990
##                               CH      : 69 1990
##                               CY      : 69 1990
##                               CZ      : 69 1990
##                               (Other):2001 (Oth
```

Funkcja `head()` wyświetla kilka pierwszych wierszy ze zbioru danych. Funkcja `summary()` wyświetla podsumowanie zbioru danych, więcej o tej funkcji napiszemy w odcinakach 11-14.

## Podsumowanie instrukcji R

W tym odcinku omawialiśmy funkcje służące do wczytywania danych z różnych formatów: tekstowych, Excelowych, binarnych. Poniżej znajduje się zestawienie wszystkich wykorzystanych w tym odcinku instrukcji.

```
## wczytywanie danych z plików tekstowych i pr
koty_ptaki <- read.table("http://biecek.pl/MOOC
                        sep=";", dec=",", heade
## wyświetlanie wartości zmiennej
koty_ptaki

## wczytywanie danych w formacie Excela za pomo
library(gdata)
koty_ptaki_excel <- read.xls("http://biecek.pl,
## wczytywanie danych w formacie binarnym za po
load(url("http://biecek.pl/MOOC/dane/koty_ptaki
## wyświetlenie zbiorów danych dostępnych w pak
data(package="PogromcyDanych")

## wczytanie danych koty_ptaki z pakietu `Pogro
library(PogromcyDanych)
koty_ptaki

## wczytanie danych z Eurostatu o wykorzystaniu
library(SmarterPoland)
tsdtr210 <- getEurostatRCV("tsdtr210")
head(tsdtr210, 3)
summary(tsdtr210)
```

---

## Zadanie

W kolejnych odcinkach będziemy korzystać ze zbioru danych o cenach aut. Ten zbiór danych można pobrać ze strony kursu w różnych formatach pod następującymi

linkami:

- dane Excelowe,  
<http://biecek.pl/MOOC/dane/auta2012mini.xls>
- dane tekstowe,  
<http://biecek.pl/MOOC/dane/auta2012mini.csv>
- dane binarne,  
<http://biecek.pl/MOOC/dane/auta2012mini.rda>
- w pakiecie `PogromcyDanych` w zmiennej `auta2012`.

*Zadanie:*

Wczytać do programu R dane wykorzystując każdy z tych formatów.

*Uwaga!* Wskazane pliki mają do 3.5 MB wielkości. Ich pobieranie może trochę potrwać jeżeli połączenie internetowe nie jest dobre lub wiele osób jednocześnie z niego korzysta.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Jak zapisać dane z programu R

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 6*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Zapisywanie danych do pliku csv](#)
- [Zapisywanie danych do pliku csv](#)
- [Zapisywanie danych tabelarycznych do plików tekstowych](#)
- [Zapisywanie danych tabelarycznych do plików w formacie Excela](#)
- [Zapisywanie danych do plików binarnych](#)
- [Podsumowanie instrukcji R](#)
- [Zadanie](#)
- [Więcej informacji](#)

## O czym jest ten odcinek

Program R jest świetnym narzędziem do przetwarzania

danych. Mając już dane przetworzone pojawia się potrzeba by wyniki gdzieś zapisać.

Podobnie jak wczytywaliśmy dane z formatów - tekstowych, binarnych lub plików Excela - możemy również zapisywać dane do różnych formatów.

W tym odcinku dowiemy się:

- jak zapisać dane tabelaryczne do plików tekstowych,
  - jak zapisać dane tabelaryczne do plików Excela (.xls lub .xlsx),
  - jak zapisać dane w formacie binarnym programu R,
  - gdzie szukać informacji o tym jak zapisać dane do innych formatów.
- 

## Zapisywanie danych do pliku csv

Zacznijmy od przykładu, w którym stworzymy sztuczny zbiór danych o trzech wierszach i dwóch kolumnach, nazwiemy go `dwie_kolumny` a następnie zapiszmy do pliku tekstowego o rozszerzeniu `csv`.

Pliki `csv` są plikami tekstowymi, ale również są poprawnie odczytywane przez program Excel, przez co jest to popularne medium wymiany danych tabelarycznych.

Aby zapisać dane do pliku możemy użyć funkcję `write.table()`. Poniższe instrukcje tworzą nowe dane oraz zapisują je do pliku `wazne_dane.csv` (w aktualnym katalogu) stosując jako separator kolumn średnik, a jako separator dziesiętny znak `.` (kropka). To, jaki katalog jest aktualnym katalogiem roboczym, można sprawdzić funkcją `getwd()`.

```
## tworzymy nowy zbiór danych
dwie_kolumny <- data.frame(litery   = c("A", "B", "C"),
                           liczby    = c(1, 2, 3))

dwie_kolumny
##      litery  liczby
## 1         A       1
## 2         B       2
## 3         C       3

write.table(dwie_kolumny, file="wazne_dane.csv",
            sep = ";", dec = ".")

## w którym katalogu się zapisało?
getwd()
## /Users/pbiecek/Documents

## Równoważnie
write.table(dwie_kolumny, file="/Users/pbiecek/Documents/wazne_dane.csv",
            sep = ";", dec = ".")
```

---

## Zapisywanie danych do pliku csv

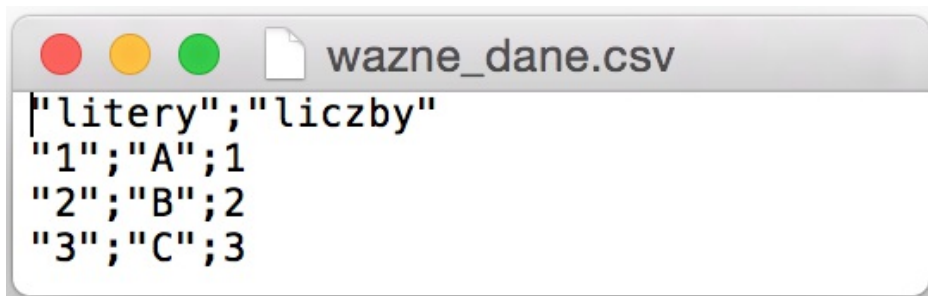


W wyniku poniższego wywołania funkcji `write.table()`

```
write.table(dwie_kolumny, file="wazne_dane.csv",  
            sep = ";", dec = ".")
```

Na dysku w aktualnym katalogu (tzw. roboczym katalogu) powstanie plik `wazne_dane.csv`.

Poniższy rysunek pokazuje treść tego pliku.



Zauważmy, że w pierwszej linii znajdują się dwie wartości - nazwy kolumn.

W kolejnych liniach występują po trzy wartości, pierwsza to nazwa wiersza a dwie kolejne to wartości pierwszej i drugiej kolumny.

---

## Zapisywanie danych tabelarycznych do plików tekstowych

Dane, które są dostępne w programie R, można zapisać do pliku tekstowego funkcją `write.table()`. Można też funkcjami `write.csv()` lub `write.csv2()`, ale różnią się one wyłącznie domyślnymi wartościami parametrów.

Formatowanie (co jest separatorem kolumn, co jest separatorem dziesiętnym, itp.) określa się w ten sam sposób co w funkcji `read.table()`.

Formalna deklaracja funkcji `write.table()` jest następująca (pamiętajmy, że większość parametrów jest domyślna i zazwyczaj wystarczy zmienić tylko kilka).

```
write.table(x, file = "", append = FALSE, quote = TRUE,
            eol = "\n", na = "NA", dec = ".",
            col.names = TRUE, qmethod = c("escape", "double"),
            fileEncoding = "")
```

Najczęściej wykorzystywane argumenty to:

- `x` - określa jaka tabela/które dane mają być zapisane do pliku tekstowego,
- `file` - określa nazwę pliku tekstowego, do którego dane będą zapisane,
- `append` - domyślna wartość `FALSE` oznacza, że jeśli plik o nazwie `file` istniał wcześniej, to zostanie on skasowany. Jeżeli ustawimy `append=TRUE` to nowe dane będą dopisane do końca poprzedniego pliku.
- `sep` - określa znak, który jest separatorem kolejnych

wartości,

- `dec` - określa znak, który jest separatorem dziesiętnym (zazwyczaj `.` lub `,`),
  - `row.names` - określa czy do pliku tekstowego mają być zapisywane nazwy wierszy, domyślnie `TRUE`,
  - `col.names` - określa czy do pliku tekstowego mają być zapisywane nazwy kolumn, domyślnie `TRUE`.
- 

## Zapisywanie danych tabelarycznych do plików w formacie Excela

Dane, które są dostępne w programie R, można zapisać też do pliku Excela funkcją `write.xls()`.

Funkcja ta jest dostępna w większości pakietów, zawierających funkcję `read.xls()`, np. w pakiecie `xlsx` (korzysta z biblioteki Java, trzeba więc mieć ją zainstalowaną) oraz `dataframe2xls` (korzysta z zewnętrznych bibliotek języka Python, trzeba więc mieć go zainstalowanego).

Do zapisu do formatu Excela można też wykorzystać funkcję `WriteXLS()` z pakietu `WriteXLS` (korzysta z zewnętrznych bibliotek języka Perl).

Podstawowa funkcjonalność tych funkcji jest taka sama,

pozwalają one na stworzenie nowego pliku oraz zapisania do niego tabeli.

Pakiety te różnią się dodatkowymi możliwościami. Niektóre potrafią zapisywać do plików Excelowych, które mają wiele zakładek. Inne potrafią zapisywać do pliku Excela z dodanym formatowaniem, takim jak określone kolory, wielkość pisma, krój czcionki, obrysowania tabeli.

Osobiście najczęściej korzystam z pakietu `xlsx`, ponieważ pozwala on na zapis danych do wielu zakładek z zadaniem formatowaniem.

```
library(xlsx)
write.xlsx(dwie_kolumny, file="/Users/pbiecek/I
          sheetName="Zakladka Wazne Dane")
```

*Uwaga* Pakiet `xlsx` trzeba przed włączeniem najpierw zainstalować poleceniem `install.packages("xlsx")`.

*Uwaga 2* Posiadacze systemu OSX w wersji Yosemite (na dzień dzisiejszy, najnowszej) mogą mieć problem z włączeniem pakietu `xlsx` jeżeli mają zainstalowaną złą (lub żadną) wersję programu Java. Należy zainstalować najnowszą wersję ze strony <http://osxdaily.com/2014/10/21/get-java-os-x-yosemite/>.

*Uwaga 3* Posiadacze systemu Ubuntu mogą doinstalować

potrzebne biblioteki poleceniem

```
sudo apt-get install r-cran-rjava
```

---

## Zapisywanie danych do plików binarnych

Do pliku w formacie natywnym R (*natywnym, czyli właściwym dla programu R*), dane można zapisać funkcją `save()`. Funkcja ta zapisuje jeden lub więcej obiektów o dowolnych strukturach - nie tylko danych tabelarycznych, ale również wektorów czy bardziej złożonych struktur (o tym czym są wektory czy inne struktury opowiemy w odcinku

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

Wadą tego rozwiązania jest to, że format ten nie jest odczytywany przez inne pakiety statystyczne.

Jeżeli jednak pracujemy głównie z programem R i chcemy w przyszłości te dane odczytywać tylko z programu R, to jest to najwygodniejsza opcja, produkująca najmniejsze pliki, które łatwo i szybko można ponownie wczytać poleceniem `load()`.

Przykład użycia funkcji `save()`.

```
## do katalogu roboczego
save(dwie_kolumny, file="wazne_dane.rda")

## podając pełną nazwę ścieżki
save(dwie_kolumny, file="/Users/pbiecek/Documents/
```

Jeżeli chcemy zapisać do pliku więcej obiektów niż jeden, wystarczy je wskazać po przecinku. W poniższym przykładzie do jednego pliku zapisane będą trzy obiekty.

```
## tworzymy dwa wektory
v1 <- 1:10
l1 <- LETTERS[1:10]
## Zapisujemy do pliku trzy obiekty: v1, l1, dwie_kolumny
save(v1, l1, dwie_kolumny, file="trzy_obiekty.rda")
```

Odczytując dane funkcją `load()`, do programu R wczytywane są obiekty razem z nazwami (pisaliśmy o tym w odcinku

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzanie\\_danych](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzanie_danych)

Dlatego możemy do jednego pliku zapisać kilka obiektów i po wczytaniu danych nie będzie kolizji nazw.

---

## Podsumowanie instrukcji R

W tym odcinku omawialiśmy funkcje służące do zapisywania danych do różnych formatów: tekstowych, Excelowych, binarnych. Poniżej znajduje się zestawienie

wszystkich wykorzystanych w tym odcinku instrukcji.

```
## tworzymy nowy zbiór danych
dwie_kolumny <- data.frame(litery = c("A", "B", "C"),
                           liczby = c(1, 2, 3))

dwie_kolumny
##      litery liczby
## 1      A      1
## 2      B      2
## 3      C      3

## zapisujemy do pliku tekstowego w lokalnym katalogu
write.table(dwie_kolumny, file="wazne_dane.csv",
            sep = ";", dec = ".")

## który katalog jest lokalny/roboczy
getwd()
## /Users/pbiecek/Documents

## Równoważnie możemy użyć ścieżki bezwzględnej
write.table(dwie_kolumny, file="/Users/pbiecek/Documents/wazne_dane.csv",
            sep = ";", dec = ".")

## zapisujemy dane do formatu Excela
library(xlsx)
write.xlsx(dwie_kolumny, file="/Users/pbiecek/Documents/wazne_dane.xlsx",
          sheetName="Zakladka Wazne Dane")

## zapisujemy dane do formatu binarnego
## do katalogu roboczego
save(dwie_kolumny, file="wazne_dane.rda")

## podając pełną nazwę ścieżki
save(dwie_kolumny, file="/Users/pbiecek/Documents/wazne_dane.rda")
```

---

# Zadanie

- Zapisz zbiór danych `koty_ptaki` do pliku tekstowego, w którym dane są rozdzielane przecinkiem a separatorem dziesiętnym jest `.` (kropka).
- Zapisz zbiór danych `koty_ptaki` do formatu pliku Excela. Sprawdź czy dane poprawnie się zapisały.
- Zapisz zbiór danych `koty_ptaki` do pliku binarnego. Odczytaj go następnie funkcją `load()` i sprawdź czy dane poprawnie zostały zapisane.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

---

## Więcej informacji

Gdzie szukać dalszych informacji o zapisie i odczycie danych w innych formatach?

Do R można wczytać i z R można zapisać dane w formacie programu SAS, SPSS, Statistica, Stata i wielu innych



- Więcej informacji o tym jak wczytywać dane z innych formatów oraz baz danych znaleźć można w rozdziale 2.2 książki „Przewodnik po pakiecie R”. Rozdział ten jest dostępny bezpłatnie pod adresem <http://biecek.pl/R>.
- Bardziej szczegółowe informacje o tym jak wczytywać i zapisywać dane z i do baz danych lub plików w egzotycznych formatach znaleźć można w dokumencie „R Data Import/Export” <http://cran.r-project.org/doc/manuals/r-release/R-data.pdf> <https://github.com/pbiecek/MOOC/blob/master/material/data.pdf?raw=true>
- O ile „R Data Import/Export” jest opisem „teoretycznym” to bardzo ciekawe zebranie przypadków użycia funkcji do wczytywania danych z różnych formatów jest zebrane w wiki-książce „R programming” dostępnej pod adresem [http://en.wikibooks.org/wiki/R\\_Programming/Importi](http://en.wikibooks.org/wiki/R_Programming/Importi)
- Interesujące zestawienie narzędzi do odczytywania danych ze stron internetowych znaleźć można na stronach z materiałami dydaktycznymi Gastona Sancheza <http://gastonsanchez.com/teaching/>

*Dla entuzjastów Excela*

- Funkcje R można również uruchamiać wewnątrz programu Excel, należy w tym celu zainstalować dodatek

RExcel(<http://www.statconn.com/products.html>). Na tych filmach można zobaczyć jak pracuje się z tym dodatkiem i jak go zainstalować

<https://www.youtube.com/watch?v=rigtfeeqnNs>,

<https://www.youtube.com/watch?v=YH94AH6PVss>,

<https://www.youtube.com/watch?v=wqgO9rrmQVY>.

# Indeksowanie ramek danych i wektorów

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 7*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Wektory](#)
- [Wektory](#)
- [Indeksowanie wektora](#)
- [Indeksowanie wektora](#)
- [Indeksowanie wektora](#)
- [Zadania](#)
- [Ptaki vs koty](#)
- [Indeksowanie wierszy w ramce danych](#)
- [Indeksowanie wierszy w ramce danych](#)
- [Indeksowanie wierszy w ramce danych](#)
- [Indeksowanie wierszy w ramce danych](#)
- [Indeksowanie kolumn w ramce danych](#)
- [Indeksowanie kolumn w ramce danych](#)
- [Indeksowanie kolumn w ramce danych](#)

- [Wybieranie podramki z ramki danych](#)
- [Wybieranie podramki z ramki danych nazwami](#)
- [Sortowanie przez indeksowanie](#)
- [Podsumowanie instrukcji R](#)
- [Podsumowanie instrukcji R](#)
- [Podsumowanie instrukcji R](#)
- [Zadania](#)

## O czym jest ten odcinek

Dane z którymi będziemy pracować mają najczęściej format tabeli lub wektora. Jedną z podstawowych operacji na tabelach oraz wektorach jest wybieranie podzbioru wierszy, kolumn lub wartości.

W tym odcinku dowiemy się:

- jak tworzyć wektory,
- jak indeksować wartości z wektora,
- jak wybierać wiersze z ramki danych,
- jak wybierać kolumny z ramki danych,
- jak wybierać wiersze i kolumny,
- jak indeksować wiersze i kolumny używając nazw lub wartości logicznych.

---

## Wektory

Jednym z podstawowych rodzajów danych w programie R są wektory.

Wektory mogą zawierać liczby, napisy, wartości logiczne lub inne typy. Zaczniemy od wektorów wartości liczbowych.

W programie R nawet jedna wartość jest wektorem, tyle że małym, jednoelementowym.

```
4
```

```
## [1] 4
```

Dłuższe wektory można tworzyć np. funkcją `c()`, która pozwala na sklejenie kilku wartości w jeden wektor.

Przykładowa instrukcja tworząca wektor o trzech elementach.

```
c(3, 4, 5)
```

```
## [1] 3 4 5
```

W programowaniu i analizie danych bardzo często wykorzystuje się wektory kolejnych wartości liczbowych. Takie wektory mają nawet specjalną nazwę: *sekwencje*.

Sekwencje kolejnych liczb można stworzyć operatorem `:`. Jeżeli chcemy zbudować sekwencję liczb równie

odległych od siebie, ale z krokiem innym niż 1, to wygodnie jest wykorzystać funkcję `seq()`.

```
2:7
```

```
## [1] 2 3 4 5 6 7
```

```
seq(from = 3, to = 15, by = 2)
```

```
## [1] 3 5 7 9 11 13 15
```

---

## Wektory

Używając funkcji `c()` można tworzyć wektory wartości logicznych (z dwoma możliwymi stanami PRAWDZA/FALSZ oznaczanymi `TRUE/FALSE`), wektory napisów oraz innego rodzaju wartości.

Aby wykorzystać wektor później, trzeba go przypisać do zmiennej. Można to zrobić używając operatora `<-` lub `=`. Jeżeli chcemy wyświetlić zawartość zmiennej to wystarczy wpisać ją do konsoli i nacisnąć ENTER.

```
co_drugi <- c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE)
co_drugi
```

```
## [1] TRUE FALSE TRUE FALSE TRUE FALSE
```

```
literki <- c("alfa", "beta", "gamma", "delta")
literki
```

```
## [1] "alfa" "beta" "gamma" "delta"
```

W dalszym ciągu tego odcinka wykorzystamy wektor kolejnych liter w języku angielskim zapisany w zmiennej `LETTERS`. Użyjemy tego wektora do ćwiczeń w indeksowaniu.

```
LETTERS
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"  
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

---

## Indeksowanie wektora

Indeksowanie oznacza wybieranie określonych wartości. Wartości wybieramy wskazując ich indeksy.

Wektor to ciąg elementów. Pierwszy z tych elementów ma indeks 1, kolejny 2, kolejny 3 i tak aż do ostatniego elementu. Indeks ostatniego elementu to jednocześnie długość wektora. Można ją sprawdzić funkcją `length()`.

```
length(LETTERS)
```

```
## [1] 26
```

Aby odwołać się do określonych indeksów wektora należy użyć operatora `[]`. Wewnątrz nawiasów kwadratowych podaje się indeks elementu, do którego

chcemy się odwołać.

```
## pierwszy element wektora  
LETTERS[1]
```

```
## [1] "A"
```

```
## piąty  
LETTERS[5]
```

```
## [1] "E"
```

```
## ostatni element wektora  
LETTERS[26]
```

```
## [1] "Z"
```

```
## zamiast podawać wartość 26 można wstawić fun  
LETTERS[length(LETTERS)]
```

```
## [1] "Z"
```

---

## Indeksowanie wektora

Odwołując się do wektorów, możemy podać indeks więcej niż jednej wartości. Zasada jest jednak taka, że indeksy muszą być wektorem. Do tego przyda nam się już poznana funkcja `c()`.

Przykładowo, aby wybrać pierwszy, piąty i ostatni element z wektora `LETTERS` musimy wpraw



skonstruować wektor z tymi trzema indeksami.

```
LETTERS[c(1, 5, 26)]
```

```
## [1] "A" "E" "Z"
```

Równoważnie można najpierw stworzyć wektor indeksów a następnie wykorzystać go do indeksowania wektora

LETTERS.

```
indeksy <- c(1, 5, 26)
LETTERS[indeksy]
```

```
## [1] "A" "E" "Z"
```

Aby wybrać więcej elementów z wektora wygodnie jest wykorzystać sekwencje.

```
## dziesięć pierwszych liter
LETTERS[1:10]
```

```
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

```
## pięć pierwszych i pięć ostatnich
LETTERS[c(1:5, 21:26)]
```

```
## [1] "A" "B" "C" "D" "E" "U" "V" "W" "X" "Y"
```

```
## co druga litera, indeksujemy sekwencją od 1
coDruga <- LETTERS[seq(from = 1, to = 26, by =
coDruga
```

```
## [1] "A" "C" "E" "G" "I" "K" "M" "O" "Q" "S"
```

---

# Indeksowanie wektora

Sekwencje wartości nie muszą być rosnące. Można je wykorzystać np. do tego by odwrócić kolejność elementów w wektorze.

```
10:1

##      [1] 10   9   8   7   6   5   4   3   2   1

LETTERS[10:1]

##      [1] "J" "I" "H" "G" "F" "E" "D" "C" "B" "A"
```

Elementy wektora można również indeksować warunkiem logicznym.

W poniższym przykładzie instrukcja `LETTERS > "K"` tworzy wektor wartości logicznych weryfikujących czy kolejna litera jest większa czy mniejsza od `K` (w porządku leksykograficznym). Taki wektor wartości logicznych można wykorzystać do indeksowania wektora `LETTERS`.

```
LETTERS > "K"

##      [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FAI
##      [12]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TI
##      [23]  TRUE  TRUE  TRUE  TRUE

## tylko litery spełniające określony warunek
## w tym przypadku litery występujące po literze
```

```
LETTERS[LETTERS > "K"]
```

```
## [1] "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U"
```

---

## Zadania

- Zbuduj sekwencję dziesięciu kolejnych małych liter alfabetu łacińskiego.
  - Zbuduj sekwencję dziesięciu kolejnych liczb nieparzystych zaczynając od 3.
  - Z wektora `LETTERS` wybierz litery na pozycjach 5, 10, 15, 20 i 25.
  - Wypisz wartości wektora `LETTERS` od końca.
- 

## Ptaki vs koty

Do pracy nad wybieraniem wierszy i kolumn wykorzystamy niewielki zbiór danych o kotach i ptakach.

Ten zbiór danych jest dostępny w pakiecie `PogromcyDanych`, wystarczy więc załadować ten pakiet. Inne sposoby na wczytanie tego zbioru danych były przedstawione w odcinku 5.

Siedem kolumn i trzynaście wierszy to dobry zbiór do ćwiczeń.

```
library(PogromcyDanych)
koty_ptaki
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	z
## 4		Puma	80.00	1.7	70	z
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Et
## 9		Strus	150.00	2.5	70	
## 10	Orzel	przedni	5.00	0.9	160	
## 11	Sokol	wedrowny	0.70	0.5	110	
## 12	Sokol	norweski	2.00	0.7	100	
## 13		Albatros	4.00	0.8	120	Po

## Indeksowanie wierszy w ramce danych

Zasadnicza różnica pomiędzy wektorem a ramką danych jest taka, że wektor jest jednowymiarowy. Wartości są ustawione w ciągu. W ramce danych wartości są ułożone w dwa wymiary, wiersze i kolumny. Indeksując wartości w ramce danych podaje się wektor indeksów dla wierszy i wektor wartości dla kolumn.

Aby odwoływać się do wierszy lub kolumn ramki danych można wykorzystać operator `[ , ]`. Obowiązkowym elementem jest przecinek. Przed przecinkiem należy wpisać indeksy wierszy, po przecinku indeksy kolumn. Jeżeli przed przecinkiem lub po przecinku nie będzie żadnych wartości to wybrane będą wszystkie elementy w wierszu / kolumnie.

Przykładowo, odwołanie się do trzeciego wiersza z ramki danych

```
koty_ptaki[3, ]
```

```
##      gatunek waga dlugosc predkosc habitat zywnosc
## 3   Jaguar  100      1.7        90  Ameryka
```

Liczbę wierszy można sprawdzić funkcją `nrow()` (odpowiednio, liczbę kolumn sprawdzamy funkcją `ncol()`).

Przydatne, jeżeli np. chcemy wyświetlić ostatni wiersz.

```
nrow(koty_ptaki)
```

```
## [1] 13
```

```
koty_ptaki[13, ]
```

```
##      gatunek waga dlugosc predkosc habitat zywnosc
## 13 Albatros   4      0.8       120  Poludnie
```

---

# Indeksowanie wierszy w ramce danych

Jeżeli chcemy wybrać więcej niż jeden wiersz należy, podobnie jak dla wektorów, podać kilka indeksów przed przecinkiem.

Aby odwołać się do kilku kolejnych wierszy można wykorzystać sekwencję zbudowaną z operatorem `:`.

Przykładowo, wiersze od 8 do 10 z ramki danych `koty_ptaki` można wyłuskać następująco.

```
koty_ptaki[8:10, ]
```

##		gatunek	waga	dlugosc	predkosc	ha
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	z
## 10	Orzel	przedni	5.00	0.9	160	1

Funkcja `c()` skleja wartości i sekwencje w wektor, który można następnie wykorzystać w indeksowaniu wierszy.

Poniższa instrukcja wyłuskuje wiersze 3, 8, 9 i 10.

```
koty_ptaki[c(3, 8:10), ]
```

##		gatunek	waga	dlugosc	predkosc	ha
## 3		Jaguar	100.00	1.7	90	Ar
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	z
## 10	Orzel	przedni	5.00	0.9	160	1

## Lub równoważnie:

```
indeksy <- c(3, 8:10)
koty_ptaki[indeksy, ]
```

##		gatunek	waga	dlugosc	predkosc	ha
## 3		Jaguar	100.00	1.7	90	Ar
## 8		Jerzyk	0.05	0.2	170	Eur
## 9		Strus	150.00	2.5	70	i
## 10		Orzel przedni	5.00	0.9	160	1

Funkcjami `head()` i `tail()` wyluskuje się kilka pierwszych lub ostatnich wierszy ze zbioru danych. Do obu funkcji można podać drugi argument, określający ile pierwszych / ostatnich wierszy chcemy odczytać, domyślnie jest to 6 wierszy.

```
head(koty_ptaki)
```

##		gatunek	waga	dlugosc	predkosc	habitat	zyw
## 1		Tygrys	300	2.5	60	Azja	
## 2		Lew	200	2.0	80	Afryka	
## 3		Jaguar	100	1.7	90	Ameryka	
## 4		Puma	80	1.7	70	Ameryka	
## 5		Leopard	70	1.4	85	Azja	
## 6		Gepard	60	1.4	115	Afryka	

```
tail(koty_ptaki)
```

##		gatunek	waga	dlugosc	predkosc	1
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	
## 10		Orzel przedni	5.00	0.9	160	
## 11		Sokol wedrowny	0.70	0.5	110	

##	12	Sokol	norweski	2.00	0.7	100	
##	13		Albatros	4.00	0.8	120	Po

## Indeksowanie wierszy w ramce danych

Jako indeksy można również wykorzystać wektor wartości logicznych.

Wybiegając trochę w przyszłość, użyjemy kolumny `predkosc` aby wybrać z ramki danych tylko te wiersze, dla których prędkość jest wyższa niż 100.

```
najszybsze <- koty_ptaki$predkosc > 100
najszybsze
```

```
##      [1] FALSE FALSE FALSE FALSE FALSE  TRUE FAI
##     [12] FALSE  TRUE
```

```
koty_ptaki[najszybsze, ]
```

##		gatunek	waga	dlugosc	predkosc	ha
##	6	Gepard	60.00	1.4	115	z
##	8	Jerzyk	0.05	0.2	170	Eu:
##	10	Orzel przedni	5.00	0.9	160	1
##	11	Sokol wedrowny	0.70	0.5	110	1
##	13	Albatros	4.00	0.8	120	Po:

## Indeksowanie wierszy w ramce danych



Indeksując wiersze lub kolumny można też wykorzystywać ujemne indeksy. Oznaczają one, wszystkie wartości poza wskazanymi.

Przykładowo, wszystkie wiersze poza 1, 3, 8, 9 i 10 można uzyskać poleceniem.

```
koty_ptaki[ -c(1, 3, 8:10), ]
```

##		gatunek	waga	dlugosc	predkosc	ha
## 2		Lew	200.0	2.0	80	z
## 4		Puma	80.0	1.7	70	Ar
## 5		Leopard	70.0	1.4	85	
## 6		Gepard	60.0	1.4	115	z
## 7		Irbis	50.0	1.3	65	
## 11	Sokol	wedrowny	0.7	0.5	110	1
## 12	Sokol	norweski	2.0	0.7	100	1
## 13		Albatros	4.0	0.8	120	Po

Zauważmy, że w tym przykładzie znak - przed funkcją c() tworzącą wektor, powoduje zmianę znaku wszystkich elementów wektora.

```
-c(1, 3, 8:10)
```

```
## [1] -1 -3 -8 -9 -10
```

Uwaga! Nie można mieszać jednocześnie indeksów dodatnich i ujemnych.



# Indeksowanie kolumn w ramce danych

Podobnie jak wiersze, można indeksować również kolumny.

Aby wyłuskać drugą kolumnę można wskazać jej numer po przecinku.

```
koty_ptaki[, 2]
```

```
##      [1] 300.00 200.00 100.00  80.00  70.00  60
##      [11]   0.70   2.00   4.00
```

Wybranie jednej kolumny powoduje, że jako wynik otrzymujemy nie ramkę danych ale wektor. Łatwo to poznać po sposobie wyświetlania danych.

Aby zapobiec takiej konwersji na wektor i jako wynik wciąż mieć ramkę danych, należy dodać do operatora indeksowania argument `drop=FALSE`.

```
koty_ptaki[,2, drop=FALSE]
```

##		waga
## 1	300.00	
## 2	200.00	
## 3	100.00	
## 4	80.00	
## 5	70.00	
## 6	60.00	
## 7	50.00	

##	8	0.05
##	9	150.00
##	10	5.00
##	11	0.70
##	12	2.00
##	13	4.00

*Uwaga* Ten dziwny zapis jest konsekwencją tego, że operator `[, ]` jest w gruncie rzeczy funkcją. Więcej o zaawansowanych elementach języka dowiemy się w kolejnych odcinkach.

---

## Indeksowanie kolumn w ramce danych

W ramce danych kolumny możemy indeksować nie tylko numerami ale również nazwami (kolumny są nazywane).

Nazwy kolumn z ramki danych można odczytać funkcją `colnames()`. Wynikiem tej funkcji jest wektor z nazwami.

```
colnames(koty_ptaki)
```

```
## [1] "gatunek"      "waga"          "dlugosc"       "pre  
## [7] "druzyna"
```

Aby wyłuskać z ramki danych kolumnę o nazwie `waga` możemy użyć tej nazwy jako indeksu.

```
koty_ptaki[, "waga"]
```

```
##      [1] 300.00 200.00 100.00 80.00 70.00 60
##      [11] 0.70 2.00 4.00
```

Ponieważ operacja odwołania się do jednej kolumny w danych jest dosyć częsta, można ją wykonać na kilka innych sposobów. Najpopularniejszym jest użycie operatora `$`.

Z jego pomocą do kolumny `waga` możemy odwołać się w ten sposób.

```
koty_ptaki$waga
```

```
##      [1] 300.00 200.00 100.00 80.00 70.00 60
##      [11] 0.70 2.00 4.00
```

---

## Indeksowanie kolumn w ramce danych

Aby wybrać więcej niż jedną kolumnę, podobnie jak w przypadku wierszy i wektorów można wykorzystać funkcję `c()`.

Przykładowo, jeżeli chcemy wybrać drugą, czwartą, piątą i szóstą kolumnę możemy użyć instrukcji.

```
## równoważnie moglibyśmy napisać
## koty_ptaki[, c("waga", "predkosc", "habitat", "rodzaj")]
koty_ptaki[, c(2,4:6)]
```

##		waga	predkosc	habitat	zywotnosc
## 1		300.00	60	Azja	25
## 2		200.00	80	Afryka	29
## 3		100.00	90	Ameryka	15
## 4		80.00	70	Ameryka	13
## 5		70.00	85	Azja	21
## 6		60.00	115	Afryka	12
## 7		50.00	65	Azja	18
## 8		0.05	170	Euroazja	20
## 9		150.00	70	Afryka	45
## 10		5.00	160	Polnoc	20
## 11		0.70	110	Polnoc	15
## 12		2.00	100	Polnoc	20
## 13		4.00	120	Poludnie	50

## Wybieranie podramki z ramki danych

Możemy jednocześnie odwoływać się do wierszy i kolumn w ramce danych, wybierając jej podramkę.

Przykładowo, wybór czterech wierszy i czterech kolumn może wyglądać tak.

```
koty_ptaki[c(3,8:10), c(2,4:6)]
```

##		waga	predkosc	habitat	zywotnosc
## 3		100.00	90	Ameryka	15
## 8		0.05	170	Euroazja	20
## 9		150.00	70	Afryka	45
## 10		5.00	160	Polnoc	20

Wybór czterech wierszy i jednej kolumny.

```
koty_ptaki[c(3,8:10), 2]
```

```
## [1] 100.00 0.05 150.00 5.00
```

Jeden wiersz i cztery kolumny.

```
koty_ptaki[3, c(2,4:6)]
```

```
##      waga predkosc habitat zywnosc
## 3    100         90 Ameryka        15
```

---

## Wybieranie podramki z ramki danych nazwami

Pokazaliśmy wcześniej, jak można odwoływać się do kolumn poprzez ich nazwy.

Podobnie można zrobić z wierszami. Funkcja `rownames()` pokazuje jak nazywają się wiersze w ramce danych.

```
rownames(koty_ptaki)
```

```
## [1] "1" "2" "3" "4" "5" "6" "7" "8"
```

Te nazwy niewiele mówią, używanie ich do indeksowania nie miałoby sensu. Zmieńmy więc nazwy wierszy na takie jak w kolumnie `gatunek`

```
rownames(koty_ptaki) <- koty_ptaki$gatunek
```

Teraz możemy odwoływać się do wierszy poprzez nazwy tych wierszy.

Poniższy przykład wybiera wiersze dla czterech wskazanych gatunków oraz trzy wybrane kolumny. Jeżeli wiersze mają sensowne nazwy, to wygodniej jest odwoływać się do wierszy przez nazwy niż przez indeksy liczbowe.

```
koty_ptaki[c("Lew", "Leopard", "Jerzyk", "Strus",  
             c("waga", "dlugosc", "predkosc")]
```

##		waga	dlugosc	predkosc
##	Lew	200.00	2.0	80
##	Leopard	70.00	1.4	85
##	Jerzyk	0.05	0.2	170
##	Strus	150.00	2.5	70

*Uwaga* Gdybyśmy chcieli pozbyć się nazw wierszy, można to zrobić instrukcją

```
rownames(koty_ptaki) <- NULL
```

---

---

## Sortowanie przez indeksowanie

Interesujące i nieco zaawansowane zastosowanie indeksowania przedstawimy na przykładzie funkcji

```
order()
```

Wynikiem tej funkcji, są indeksy kolejnych, rosnących wartości.

Przykładowo, w kolumnie `predkosc` mamy następujące wartości

```
koty ptaki[, "predkosc"]
```

```
##      [1] 60 80 90 70 85 115 65 170 70 160
```

Wynikiem funkcji `order()` są indeksy kolejnych, wartości. Najmniejsza wartość to 60, na pozycji 1, kolejna to 65 na pozycji 7, kolejna to 70 na pozycjach 4 i 9 i tak dalej.

```
order(koty_ptaki[, "predkosc"])
```

```
##      [1]      1      7      4      9      2      5      3     12     11      6     13     10      8
```

Możemy wykorzystać ten wynik, aby posortować ramkę danych po określonej kolumnie. W przykładzie poniżej wykorzystujemy funkcję `order()` do wyznaczenia wektora `kolejnosc`. Który następnie wykorzystamy do indeksowania ramki `koty_ptaki`.

```
kolejnosc <- order(koty_ptaki[, "predkosc"])
koty_ptaki[kolejnosc, ]
```

```
##          gatunek    waga  dlugosc
```



##	Tygrys	Tygrys	300.00	2.5
##	Irbis	Irbis	50.00	1.5
##	Puma	Puma	80.00	1.7
##	Strus	Strus	150.00	2.5
##	Lew	Lew	200.00	2.0
##	Leopard	Leopard	70.00	1.4
##	Jaguar	Jaguar	100.00	1.7
##	Sokol norweski	Sokol norweski	2.00	0.7
##	Sokol wedrownny	Sokol wedrownny	0.70	0.5
##	Gepard	Gepard	60.00	1.4
##	Albatros	Albatros	4.00	0.8
##	Orzel przedni	Orzel przedni	5.00	0.9
##	Jerzyk	Jerzyk	0.05	0.2
##	druzyna			
##	Tygrys	Kot		
##	Irbis	Kot		
##	Puma	Kot		
##	Strus	Ptak		
##	Lew	Kot		
##	Leopard	Kot		
##	Jaguar	Kot		
##	Sokol norweski	Ptak		
##	Sokol wedrownny	Ptak		
##	Gepard	Kot		
##	Albatros	Ptak		
##	Orzel przedni	Ptak		
##	Jerzyk	Ptak		

## Podsumowanie instrukcji R

W tym odcinku omawialiśmy funkcje służące do tworzenia wektorów, sekwencji oraz indeksowania wektorów.

```
## tworzenie wektorów liczb i sekwencji
c(3, 4, 5)
2:7

seq(from = 3, to = 15, by = 2)

## wektory wartości logicznych i napisowych
co_drugi <- c(TRUE, FALSE, TRUE, FALSE, TRUE, 1
co_drugi

literki <- c("alfa", "beta", "gamma", "delta")
literki

## wyznaczanie długości wektora LETTERS
length(LETTERS)

## indeksowanie pojedynczego elementu wektora,
LETTERS[1]

## zamiast podawać wartość 26 można wstawić fun
LETTERS[length(LETTERS)]

## indeksowanie kilku elementów wektora
LETTERS[c(1,5,26)]

## z użyciem zmiennej pomocniczej
indeksy <- c(1, 5, 26)
LETTERS[indeksy]

## dziesięć pierwszych liter od J do A
LETTERS[10:1]

## pięć pierwszych i pięć ostatnich
LETTERS[c(1:5, 21:26)]

## co druga litera, indeksujemy sekwencją od 1
LETTERS[seq(from = 1, to = 26, by = 2)]

## używanie wartości logicznych do indeksowania
```

---

## Podsumowanie instrukcji R

W tym odcinku omawialiśmy funkcje służące do indeksowania wierszy w ramkach danych.

```
## indeksowanie pojedynczego wiersza
koty_ptaki[3, ]

## liczba wierszy
nrow(koty_ptaki)

## indeksowanie kilku wierszy
koty_ptaki[8:10, ]
koty_ptaki[c(3, 8:10), ]
## z użyciem zmiennej pomocniczej
indeksy <- c(3, 8:10)
koty_ptaki[indeksy, ]

## pierwsze 6 wierszy i ostatnie 6 wierszy
head(koty_ptaki)
tail(koty_ptaki)

## indeksowanie wierszy warunkiem logicznym
najszybsze <- koty_ptaki$predkosc > 100
koty_ptaki[najszybsze, ]

## używanie ujemnych indeksów by pominąć wiersz
koty_ptaki[ -c(1, 3, 8:10), ]
```

---

# Podsumowanie instrukcji R

W tym odcinku omawialiśmy funkcje służące do indeksowania wierszy, kolumn oraz jednocześnie wierszy i kolumn.

```
## indeksowanie kolumn, tylko druga kolumna, wiersze 3-10
koty_ptaki[, 2]
## drop=FALSE powoduje, że wynikiem jest ramka danych
koty_ptaki[,2, drop=FALSE]

## nazwy kolumn
colnames(koty_ptaki)

## indeksowanie jednej kolumny z użyciem nazwy
koty_ptaki[, "waga"]
koty_ptaki$waga

## indeksowanie kilku kolumn
## koty_ptaki[, c("waga", "predkosc", "habitat")]
koty_ptaki[, c(2,4:6)]

## jednoczesne indeksowanie wierszy i kolumn
koty_ptaki[c(3,8:10), c(2,4:6)]

## wektor nazw wierszy
rownames(koty_ptaki)
## przypisanie nowych nazw dla wierszy
rownames(koty_ptaki) <- koty_ptaki$gatunek

## odwoływanie się przez nazwy wierszy i kolumn
koty_ptaki[c("Lew", "Leopard", "Jerzyk", "Struszy", "Kot", "Koczkodło", "Kot", "Koczkodło", "Kot", "Koczkodło"),
            c("waga", "dlugosc", "predkosc")]
```

# Zadania

- Wybierz z ramki danych `koty_ptaki` wszystkie wiersze poza „Sokołami” (wiersz 11 i 12).
- Wybierz z ramki danych `koty_ptaki` tylko koty (pierwsze siedem wierszy).
- Wybierz z ramki danych `koty_ptaki` tylko kolumnę z wagą i prędkością.
- Wybierz z ramki danych `koty_ptaki` wszystkie kolumny poza ostatnią.
- Wybierz z ramki wiersze dla których waga jest mniejsza niż 100 oraz cztery pierwsze kolumny.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Pętle

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 8*

*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Pętle - od 1 do 5](#)
- [Pętle - schemat](#)
- [Pętle - schemat](#)
- [Pętle - koszty napraw](#)
- [Pętle - koszty napraw](#)
- [Pętle - wartości w sekwencji](#)
- [Pętle - nazwy kolumn](#)
- [Pętle - nazwy kolumn](#)
- [Pętle - iteracja po wierszu / kolumnie](#)
- [Pętle - iteracja po wierszu / kolumnie](#)
- [Wektoryzacja](#)
- [Pętla w pętli](#)
- [Pętla w pętli](#)
- [Pętla w pętli](#)
- [Podsumowanie instrukcji R](#)
- [Podsumowanie instrukcji R](#)
- [Zadania](#)

# O czym jest ten odcinek

Silną stroną komputerów jest to, że mogą bardzo szybko powtarzać określone operacje. Nie są zbyt inteligentne (póki co), ale są bardzo szybkie i idealne do powtarzania, powtarzania, powtarzania, powtarzania...

Analizując dane bardzo często znajdziemy się w sytuacji, gdy określoną czynność będziemy chcieli powtórzyć. Wykonać ją dla każdej zmiennej, dla każdego analizowanego okresu czasu, dla każdego analizowanego zbioru danych.

Narzędziem pozwalającym na powtarzanie w kółko (w pętli) określonych operacji są właśnie... pętle.

W tym odcinku nauczymy się:

- jak i po co tworzyć pętle,
- jak korzystać z pętli `for()`,
- jak i po co tworzyć pętle w pętli.

Do ilustracji wykorzystamy zbiór danych `koty_ptaki` dostępny w pakiecie `PogromcyDanych`.

```
library(PogromcyDanych)
head(koty_ptaki, 3)
```

```
##          gatunek waga dlugosc predkosc habitat
```

##	Tygrys	Tygrys	300	2.5	60	Azja
##	Lew	Lew	200	2.0	80	Afryka
##	Jaguar	Jaguar	100	1.7	90	Ameryka

---

## Pętle - od 1 do 5

Pętle mogą być wykorzystywane, aby powtórzyć jakąś operację określoną liczbę razy.

Zacznijmy od przykładu, w którym wielokrotnie wyświetlimy pewien napis na ekranie. W języku R jest kilka rodzajów pętli, które można do tego celu wykorzystać, ale najczęściej stosowana jest pętla `for` i to ją przedstawimy w pierwszej kolejności (znak `\n` powoduje przejście do nowej linii).

```
odliczajDo <- 5

for (i in 1:odliczajDo) {
  cat("Wartość zmiennej i: ", i, "\n")
}
```

```
## Wartość zmiennej i: 1
## Wartość zmiennej i: 2
## Wartość zmiennej i: 3
## Wartość zmiennej i: 4
## Wartość zmiennej i: 5
```

Ten sam efekt otrzymalibyśmy wpisując pięciokrotnie instrukcję `cat()` (która wyświetla swoje argumenty na



ekranie).

```
cat("Wartość zmiennej i: 1\n")
cat("Wartość zmiennej i: 2\n")
cat("Wartość zmiennej i: 3\n")
cat("Wartość zmiennej i: 4\n")
cat("Wartość zmiennej i: 5\n")
```

Co jednak, gdybyśmy chcieli powtórzyć ją 10 000 razy? Z pewnością nikt z nas nie chciałby wpisywać przez kilka dni tych instrukcji, tym bardziej, że ten sam efekt można osiągnąć prostą pętlą.

---

## Pętle - schemat

Sposób użycia (tak zwana *składnia*) pętli `for` jest następująca:

```
for (A in B) {
  C
}
```

W wyniku działania tej pętli wyrażenie `C` będzie wykonane dla każdej wartości z wektora `B`. Co więcej, aktualny krok pętli jest określony przez zmienną `A` i można z tej wartości korzystać wewnątrz wyrażenia `C`.

W przykładzie z poprzedniego slajdu wyrażenie

`cat("Wartość zmiennej i: ", i, "\n")` jest wykonywane dla każdego elementu wektora `1:odliczajDo`. W każdym kroku pętli zmienna `i` przyjmuje kolejną wartość z wektora `1:odliczajDo`.

```
for (i in 1:odliczajDo) {  
  cat("Wartość zmiennej i: ", i, "\n")  
}
```

## Pętle - schemat

Poniższy schemat ilustruje sposób działania pętli `for`.

### Pętla **for**

Dla każdej wartości w wektorze **B** podstaw tę wartość do zmiennej **A** i wykonaj instrukcje w bloku **C**

```
for (A in B) {  
  C  
}
```

Przykładowe wartości w **B**=`c(1, 2, 3, 4, 5)`

- 
- 1 ustaw **A** = 1  
wykonaj instrukcje w **C**
  - 2 ustaw **A** = 2  
wykonaj instrukcje w **C**
  - 3 ustaw **A** = 3  
wykonaj instrukcje w **C**
  - 4 ustaw **A** = 4  
wykonaj instrukcje w **C**
  - 5 ustaw **A** = 5  
wykonaj instrukcje w **C**

# Pętle - koszty napraw

W poprzednim przykładzie łatwo było przewidzieć wynik całej pętli. Wyświetlała ona pięć kolejnych liczb od 1 do 5.

Pokażemy teraz bardziej złożony przykład, który symuluje tak zwany proces szkodowy. Założmy, że mamy flotę samochodów, w której w ciągu roku zdarzyć się może  $X$  stłuczek ( $X$  będziemy losować). Naprawa każdej stłuczki kosztować będzie od 0 do 10 000 PLN.

```
## losujemy liczbę stłuczek
liczbaStluczek <- round(runif(n = 1, min = 1, r
## dla każdej losujemy wartość szkody
for (i in 1:liczbaStluczek) {
  cat("Stłuczka ", i, ", a jej koszt naprawy to
      runif(n = 1, min=0, max=10000), " PLN\n",
}
```

```
## Stłuczka 1, a jej koszt naprawy to 3733.065
## Stłuczka 2, a jej koszt naprawy to 4916.305
```

W tym przykładzie funkcja `runif()` służy do losowania  $n$  liczb z przedziału od `min` do `max`. W pierwszej linii wykorzystujemy ją do wylosowania liczby stłuczek z przedziału 1 do 10. Funkcja `round()` służy do zaokrąglania wyniku do liczby całkowitej.

W każdym kroku pętli losowana jest kolejna wartość, oraz

wypisywana jest na ekranie. Choć ta pętla wygląda na prostą i niewinną, podobny sposób losowania wielkości szkody jest w rzeczywistości wykorzystywany do zarządzania niepewnością dotyczącego tego, jak duże szkody z tytułu stłuczek lub innych losowych wypadków mogą wystąpić w firmach.

---

## Pętle - koszty napraw

W pętli z poprzedniego przykładu wyświetlaliśmy na ekranie kolejne koszty. A co trzeba zrobić by te wszystkie koszty zliczyć?

Wewnątrz pętli możemy wykonywać operacje na liczbach. Przykładowo możemy te koszty zsumować.

Wewnątrz pętli w poniższym przykładzie co krok zmienia się wartość `sumaKosztow`. Instrukcja

```
sumaKosztow <- sumaKosztow + koszt
```

 powoduje, że wartość zmiennej `sumaKosztow` zostanie zastąpiona sumą `sumaKosztow` i `koszt`.

```
## losujemy liczbę stłuczek
liczbaStluczek <- round(runif(n = 1, min = 1, r
sumaKosztow <- 0
## dla każdej losujemy wartość szkody
for (i in 1:liczbaStluczek) {
  koszt <- runif(n = 1, min=0, max=10000)
```

```
sumaKosztow <- sumaKosztow + koszt
cat("Stłuczka ", i, ", a jej koszt naprawy to ", koszt, "\n")
cat("Suma dotychczasowych kosztów to ", sumaKosztow, "\n")
}
```

```
## Stłuczka 1, a jej koszt naprawy to 3544.4 PLN
## Suma dotychczasowych kosztów to 3544.4 PLN
## Stłuczka 2, a jej koszt naprawy to 7260.361 PLN
## Suma dotychczasowych kosztów to 10804.76 PLN
## Stłuczka 3, a jej koszt naprawy to 2405.081 PLN
## Suma dotychczasowych kosztów to 13209.84 PLN
## Stłuczka 4, a jej koszt naprawy to 4524.22 PLN
## Suma dotychczasowych kosztów to 17734.06 PLN
## Stłuczka 5, a jej koszt naprawy to 9938.341 PLN
## Suma dotychczasowych kosztów to 27672.4 PLN
```

```
sumaKosztow
```

```
## [1] 27672.4
```

---

## Pętle - wartości w sekwencji

Przedstawione jak dotąd pętle przechodziły po wartościach od 1 do określonej liczby. Jest tak bardzo często, ale nie zawsze.

Indeksy pętli mogą być dowolnymi wartościami, np. liczbami z krokiem co 3. Pokażmy na przykładzie, w którym wyświetlimy co trzecią nazwę zwierzęcia z tabeli `koty_ptaki`.

```
coTrzeci <- seq(1, nrow(koty_ptaki), 3)
coTrzeci
```

```
## [1] 1 4 7 10 13
```

```
for (i in coTrzeci) {
  # aby się dobrze wyświetlała, i-tą wartość z
  nazwa <- as.character(koty_ptaki[i, "gatunek"]
  cat("Zwierzak ", i, " to ", nazwa, "\n", sep
}
```

```
## Zwierzak 1 to Tygrys
## Zwierzak 4 to Puma
## Zwierzak 7 to Irbis
## Zwierzak 10 to Orzel przedni
## Zwierzak 13 to Albatros
```

---

## Pętle - nazwy kolumn

Jak widzieliśmy w poprzednim przypadku, liczba powtórzeń pętli nie musi być z góry określona, może zależeć od np. liczby wierszy lub liczby kolumn w ramce danych.

W poniższym przykładzie dla każdej kolumny z ramki danych `koty_ptaki` wyświetlamy listę różnych wartości, które występują w tej kolumnie. Na kolejnym slajdzie wyjaśnimy krok po kroku co dzieje się w poniższym kodzie.

```
## wektor z nazwami kolumn
kolumny <- colnames(koty_ptaki)
kolumny

## [1] "gatunek"      "waga"          "dlugosc"       "predkosc"
## [7] "druzyna"

## pętla, która dla każdej kolumny wyświetla różnice
for (i in kolumny) {
  wartosciWKolumnie <- unique(as.character(koty_ptaki[,i]))
  cat("Kolumna", i, "\n")
  cat("    ", length(wartosciWKolumnie), "różnych wartości\n")
}

## Kolumna gatunek
##      13 różnych wartości: Tygrys Lew Jaguar
## Kolumna waga
##      13 różnych wartości: 300 200 100 80 70 60 50 40 30 20 10 5 2
## Kolumna dlugosc
##      10 różnych wartości: 2.5 2 1.7 1.4 1.3 1.2 1.1 1 0.9 0.8
## Kolumna predkosc
##      12 różnych wartości: 60 80 90 70 85 115 100 120 130 140 150 160
## Kolumna habitat
##      6 różnych wartości: Azja Afryka Ameryka Australia Antarktyda
## Kolumna zywnosc
##      10 różnych wartości: 25 29 15 13 21 12 11 10 9 8
## Kolumna druzyna
##      2 różnych wartości: Kot Ptak
```

## Pętle - nazwy kolumn

Przyjrzyjmy się pętli z poprzedniego slajdu linia po linii.

Pierwsza instrukcja wykorzystuje funkcję `colnames()` do wyznaczenia nazw kolumn. Te nazwy zostaną przypisane do wektora `kolumny`, który następnie zostanie wyświetlony.

```
## wektor z nazwami kolumn
kolumny <- colnames(koty_ptaki)
kolumny
```

Instrukcja `for()` rozpoczyna pętlę. Tym razem zmienna `i` będzie przebiegała po nazwach kolumn. Dla każdej nazwy kolumny wykona się ciało pętli.

```
## pętla, która dla każdej kolumny wyświetla r
for (i in kolumny) {
```

Wewnątrz pętli wyznaczany jest wektor różnych wartości. Jest to wykonywane w jednej linii, choć potrzebne do tego jest kilka instrukcji. Punktem wyjściowym jest wektor `koty_ptaki[,i]`. Używając funkcji `as.character()` wartości te (dla niektórych kolumn liczby dla innych napisy) są przekształcane na napisy. Następnie funkcja `unique()` usuwa powtarzające się elementy.

```
wartosciWKolumnie <- unique(as.character(koty
```

Dwie kolejne instrukcje `cat()` wyświetlają na ekranie w pierwszej linii nazwę kolumny, a w drugiej liczbę różnych wartości (długość wektora `wartosciWKolumnie` wyznaczana jest funkcją `length()`) oraz wszystkie różne



wartości.

```
cat("Kolumna", i, "\n")
cat("      ", length(wartosciWKolumnie), "różnych")
}
```

---

## Pętle - iteracja po wierszu / kolumnie

Typowe zastosowania pętli to wykonanie pewnej operacji dla każdego wiersza lub każdej kolumny zbioru danych.

Pokażmy teraz przykład wykorzystania pętli dla każdego wiersza, a na początek dla wiersza o indeksie 1. Przy czym tę wartość przypiszemy do zmiennej o nazwie `i` (może być też dowolna inna nazwa), a następnie wykorzystamy tę zmienną do indeksowania zbioru `koty_ptaki`.

W poniższym przykładzie będziemy chcieli wyświetlić informację do jakiej wagi dochodzą przedstawiciele danego gatunku.

```
## Zamiana zmiennej factor na zmienną napisową
koty_ptaki$gatunek <- as.character(koty_ptaki$gatunek)

## wykonamy instrukcję dla pierwszego wiersza,
i <- 1
cat(koty_ptaki[i, "gatunek"], "może ważyć do",
```

```
## Tygrys może ważyć do 300 kg
```

---

## Pętle - iteracja po wierszu / kolumnie

Powtórzmy tę operację dla każdego wiersza. Liczbę wierszy w ramce danych można odczytać poleceniem `nrow()` (od *number of rows*).

Sekwencja `1:nrow(koty_ptaki)` tworzy wektor liczb od 1 do 13 (tyle jest wierszy). Dla każdego wiersza uruchamiamy instrukcję z poprzedniego slajdu, wypisującą wagę kolejnego gatunku.

```
for (i in 1:nrow(koty_ptaki)) {  
  cat(koty_ptaki[i,"gatunek"], "może ważyć do",  
)
```

```
## Tygrys może ważyć do 300 kg  
## Lew może ważyć do 200 kg  
## Jaguar może ważyć do 100 kg  
## Puma może ważyć do 80 kg  
## Leopard może ważyć do 70 kg  
## Gepard może ważyć do 60 kg  
## Irbis może ważyć do 50 kg  
## Jerzyk może ważyć do 0.05 kg  
## Strus może ważyć do 150 kg  
## Orzeł przedni może ważyć do 5 kg  
## Sokół wędrowny może ważyć do 0.7 kg  
## Sokół norweski może ważyć do 2 kg  
## Albatros może ważyć do 4 kg
```

---

# Wektoryzacja

Bardzo często pętle można unikać, ponieważ program R w naturalny sposób pracuje na wektorach.

Przykładowo, efekt identyczny do pętli z poprzedniego slajdu można uzyskać pracując bezpośrednio na wektorach kolumnowych. Zarówno `koty_ptaki[, "gatunek"]` jak i `koty_ptaki[, "waga"]` to wektory, funkcja `paste()` połączy je w wektor napisów a funkcja `cat()` wypisze wszystkie elementy tego wektora.

```
cat(paste(koty_ptaki[, "gatunek"], "może ważyć do 300 kg"))
```

```
## Tygrys może ważyć do 300 kg
## Lew może ważyć do 200 kg
## Jaguar może ważyć do 100 kg
## Puma może ważyć do 80 kg
## Leopard może ważyć do 70 kg
## Gepard może ważyć do 60 kg
## Irbis może ważyć do 50 kg
## Jerzyk może ważyć do 0.05 kg
## Strus może ważyć do 150 kg
## Orzeł przedni może ważyć do 5 kg
## Sokół wędrowny może ważyć do 0.7 kg
## Sokół norweski może ważyć do 2 kg
## Albatros może ważyć do 4 kg
```

W języku R, jak i w każdym dojrzałym języku programowania, ten sam efekt można uzyskać na wiele

sposobów.

---

## Pętla w pętli

Pętle można zanurzać w innych pętlach. Na pierwszy rzut oka może wyglądać to dziwnie, ale jest to dosyć popularna praktyka.

Przypuśćmy, że chcemy policzyć coś, np. korelację, podobieństwo lub różnice, dla każdej pary kolumn. Jak to zrobić? Najłatwiej użyć dwóch pętli - jednej w drugiej.

```
for (i in colnames(koty_ptaki)) {  
  for (j in colnames(koty_ptaki)) {  
## poniższa instrukcja wykona się dla każdej k  
    cat("Kolumna '", i, "' i kolumna '", j, "  
  }  
}
```

---

## Pętla w pętli

Wykorzystajmy możliwość tworzenia pętli w pętli do tekstowej wizualizacji danych o prędkości zwierząt.

Zacznijmy od jednej pętli, która przedstawi prędkość za pomocą kropek. Im większa prędkość, tym więcej kropek.

Każde 5 km/h przedstawmy za pomocą kropki na ekranie.

Pierwszy wiersz w danych `koty_ptaki` opisuje prędkość Sokoła norweskiego. Odczytajmy ją (polecenie `koty_ptaki[i, "predkosc"]`), policzmy ile kropek jest potrzebnych by ją przedstawić (w tym celu dzielimy na 5) i narysujmy te kropki w pętli używając instrukcji `cat()`.

```
## używamy pętli by rysować koty_ptaki[i, "predkosc"]  
i <- 1  
n_kropek <- koty_ptaki[i, "predkosc"] / 5  
for (i in 1:n_kropek) {  
  cat(".")  
}
```

```
## .....
```

---

## Pętla w pętli

Powtórzmy tę pętlę dla każdego wiersza danych `koty_ptaki`, a więc różnych gatunków.

Powtórzmy powyższą pętlę przedstawiającą prędkość pierwszego gatunku, ale tym razem dla każdego wiersza osobno. Kropkami przedstawmy prędkość na końcu dopisując nazwę gatunku.

```
## Dla każdego wiersza w tabeli `koty_ptaki`  
for (i in 1:nrow(koty_ptaki)) {
```

```

n_kropek <- koty_ptaki[i,"predkosc"] / 5
# rysowanie kropek
for (j in 1:n_kropek) {
  cat(".")
}
# nazwa gatunku
cat(" ", koty_ptaki[i,"gatunek"], "\n")
}

## ..... Tygrys
## ..... Lew
## ..... Jaguar
## ..... Puma
## ..... Leopard
## ..... Gepard
## ..... Irbis
## ..... Jerzyk
## ..... Strus
## ..... Orzeł przelotny
## ..... Sokół wędrowny
## ..... Sokół norweski
## ..... Albatros

```

---

## Podsumowanie instrukcji R

W tym odcinku omawialiśmy pętlę `for`. Jest to najpopularniejsza z pętli. Po poznaniu funkcji w odcinku 10 poznamy też inne pętle o podobnych możliwościach co `for()` ale krótszym / wygodniejszym zapisie.

Poniżej znajduje się zestawienie wszystkich

wykorzystanych w tym odcinku instrukcji.

```
## pętla odliczająca od 1 do 5
odliczajDo <- 5
for (i in 1:odliczajDo) {
  cat("Wartość zmiennej i: ", i, "\n")
}

## pętla losująca wartości stłuczek dla losowej
liczbaStluczek <- round(runif(n = 1, min = 1, max = 10))
## dla każdej losujemy wartość szkody
for (i in 1:liczbaStluczek) {
  cat("Stłuczka ", i, " a jej koszt naprawy to", round(runif(1, min = 1, max = 1000)), "\n")
}

## pętla wypisująca liczbę różnych wartości i i j
kolumny <- colnames(koty_ptaki)
for (i in kolumny) {
  wartosciWKolumnie <- unique(as.character(koty_ptaki[, i]))
  cat("Kolumna", i, "\n")
  cat("    ", length(wartosciWKolumnie), "różnych wartości\n")
}
```

---

## Podsumowanie instrukcji R

Poniżej znajduje się zestawienie wszystkich wykorzystanych w tym odcinku instrukcji.

```
## pętla wypisująca dla każdego wiersza ciężar
for (i in 1:nrow(koty_ptaki)) {
  cat(koty_ptaki[i, "gatunek"], "może ważyć do", round(runif(1, min = 1, max = 1000)), "\n")
}
```

```
## używając operacji na wektorach czasem można
cat(paste(koty_ptaki[, "gatunek"], "może ważyć c

## pętla w pętli pozwala na wykonanie pewnej in
for (i in colnames(koty_ptaki)) {
  for (j in colnames(koty_ptaki)) {
    cat("Kolumna '", i, "' i kolumna '", j, "'
  }
}

## używając pętli w pętli możemy też stworzyć i
for (i in 1:nrow(koty_ptaki)) {
  n_kropek <- koty_ptaki[i, "predkosc"] / 5
  # rysowanie kropek
  for (j in 1:n_kropek) {
    cat(".")
  }
  # nazwa gatunku
  cat(" ", koty_ptaki[i, "gatunek"], "\n")
}
```

---

## Zadania

- Napisz pętlę, która dla każdego wiersza z tabeli `koty_ptaki` wypisze żywotność określonego gatunku.
- Napisz pętlę, która przedstawi żywotność za pomocą wykresu, na którym każdy rok przedstawiony jest jako jeden #.



- Napisz pętlę, która narysuje wykres z żywotnością, ale z nazwami gatunku po lewej stronie. Co więcej, przed każdą nazwą gatunku należy dodać tyle znaków spacja, aby nazwy były wyrównane do prawej strony. Długość napisu, mierzoną w liczbie znaków można odczytać funkcją `nchar()`.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

# Instrukcje warunkowe

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 9*

*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Instrukcje warunkowe](#)
- [Instrukcje warunkowe](#)
- [Instrukcje warunkowe](#)
- [Instrukcje warunkowe](#)
- [Pętle i instrukcje warunkowe](#)
- [Pętle i instrukcje warunkowe](#)
- [Podsumowanie instrukcji R](#)
- [Podsumowanie instrukcji R](#)
- [Zadania](#)

## O czym jest ten odcinek

Mnogość dostępnych algorytmów do analizy danych stawia nas często przed koniecznością wyboru. Czy użyć metody A, czy metody B? Każda z nich może być lepsza w innej sytuacji. Najlepszym rozwiązaniem jest często

warunkowe użycie jednej lub drugiej metody w zależności od tego jak wyglądają nasze dane.

Do warunkowego wykonania jednej lub drugiej grupy instrukcji służą instrukcje warunkowe.

W tym odcinku nauczymy się:

- jak korzystać z instrukcji warunkowych,
- jak korzystać z instrukcji warunkowych w pętli.

Do ilustracji wykorzystamy zbiór danych `koty_ptaki` dostępny w pakiecie `PogromcyDanych`.

```
library(PogromcyDanych)
head(koty_ptaki, 3)
```

##		gatunek	waga	dlugosc	predkosc	habitat
##	Tygrys	Tygrys	300	2.5	60	Azja
##	Lew	Lew	200	2.0	80	Afryka
##	Jaguar	Jaguar	100	1.7	90	Ameryka

---

## Instrukcje warunkowe

Przygotowując program często jesteśmy w sytuacji, w której dalsza akcja zależy od pewnej wartości, dla nas nieznanej lub potencjalnie zmieniającej się.

Przykładowo, jeżeli losujemy liczbę z przedziału 0-1 i w zależności od tego jaka liczba została wylosowana chcemy napisać `Orzeł` lub `Reszka`, w czasie pisania kodu nie wiemy jaka wartość się wylosuje.

Używając instrukcji warunkowych możemy opisać alternatywne scenariusze, które będą wykonane w zależności od określonego warunku.

Poniższy kod wypisze na ekranie napis `Orzeł`, jeżeli wylosuje się liczba mniejsza od 0.5 oraz napis `Reszka`, jeżeli wylosuje się liczba większa lub równa 0.5. Wylosować liczbę z przedziału 0-1 można funkcją `runif()`.

```
liczbaLosowa <- runif(n = 1)

if (liczbaLosowa < 0.5) {
  cat("Orzeł")
} else {
  cat("Reszka")
}

## Orzeł
```

---

## Instrukcje warunkowe

Przygotowując algorytm przetwarzania danych często

potrzebujemy rozważyć różne możliwe sytuacje. Aby zapisać warianty wykonywania w zależności od stanu określonych zmiennych, można wykorzystać instrukcje warunkowe.

Najczęściej wykorzystywaną instrukcją warunkową jest `if else`, która w języku R występuje w trzech odmianach.

Jeżeli `warunek` występujący po bloku `if` jest prawdziwy, wykonana będzie instrukcja `wyrażenie1`, w przeciwnym przypadku `wyrażenie2` (o ile jest wskazane).

Trzecia wymieniona poniżej odmiana to instrukcja `ifelse()`, która pracuje nie na jednej wartości logicznej TRUE/FALSE ale na całym wektorze wartości logicznych. Jej wynikiem jest również wektor wartości.

```
if (warunek)
  wyrażenie1
```

```
if (warunek)
  wyrażenie1
else
  wyrażenie2
```

```
ifelse (warunek_dla_wektora, wyrażenie1, wyrażenie2)
```

Jeżeli wyrażenie obejmuje więcej niż jedną instrukcję, należy je otoczyć blokiem `{ }`.

Nawet gdy wyrażenie jest jedną instrukcją, to dodawanie nawiasów klamrowych {} jest popularną praktyką, powoduje bowiem, że kod jest czytelniejszy.

```
if (warunek) {
    wyrażenie1
}

if (warunek) {
    wyrażenie1
} else {
    wyrażenie2
}
```

## Instrukcje warunkowe

Zauważmy jeszcze, że w kolumnie gatunek dane są przechowywane nie jako napisy ale jako zmienne jakościowe (więcej o tych zmiennych napiszemy w odcinku 12. Cechy jakościowe).

```
koty_ptaki$gatunek

##      [1] "Tygrys"          "Lew"              "Jagu
##      [5] "Leopard"         "Gepard"           "Irb:
##      [9] "Strus"           "Orzel przedni"    "Sok
##     [13] "Albatros"
```

Dlatego zanim przystąpimy do pracy, zastąpimy te zmienne napisami (zrobi to funkcja as.character()).

```
koty_ptaki$gatunek <- as.character(koty_ptaki$gatunek)
```

Na przykładzie dziewiątego wiersza z tabeli danych `koty_ptaki` zademonstrujemy jak działają kolejne odmiany instrukcji warunkowej. Kolumnę `gatunek` musimy zamienić na kolumnę napisów, inaczej nazwy gatunków będą się źle wyświetlały. Będziemy pracować na dziewiątym wierszu, ale indeks tego wiersza zapiszemy w zmiennej `i`. Dzięki temu, łatwiej nam będzie później wykorzystać ten kod w pętli.

```
i <- 9
koty_ptaki[i,]

##          gatunek waga dlugosc predkosc habitat
## Strus      Strus  150      2.5         70  Afryka
```

Jeżeli użyjemy wyłącznie bloku `if`, to w sytuacji gdy warunek jest fałszywy nic się nie wyświetli na ekranie.

```
if (koty_ptaki[i,"druzyna"] == "Kot") {
  cat(koty_ptaki[i,"gatunek"], "to duży kot.")
}
```

Jeżeli użyjemy instrukcji `if-else`, to gdy warunek okaże się być fałszywy wyświetli się wynik części `else`.

```
if (koty_ptaki[i,"druzyna"] == "Kot") {
  cat(koty_ptaki[i,"gatunek"], "to duży kot.")
} else {
  cat(koty_ptaki[i,"gatunek"], "to ptak.")
}
```

```
}  
  
## Strus to ptak.
```

---

## Instrukcje warunkowe

Jeżeli pracujemy na wektorach, wygodnym rozwiązaniem jest skorzystanie z instrukcji wektorowej `ifelse()`.

Jej pierwszym argumentem może być wektor.

Przykładowo instrukcja

`koty_ptaki[, "druzyna"] == "Kot"` tworzy wektor wartości logicznych.

```
koty_ptaki[, "druzyna"] == "Kot"
```

```
##      [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TI  
##      [12] FALSE FALSE
```

Wykorzystamy ją w instrukcji `ifelse()` i jako wynik otrzymamy wektor napisów o wartościach

"Kolejny wielki kot" tam gdzie występowała wartość TRUE i wartościach "Kolejny szybki ptak" tam gdzie występowała wartość FALSE.

```
ifelse(koty_ptaki[, "druzyna"] == "Kot",  
       "Kolejny wielki kot",  
       "Kolejny szybki ptak")
```



```
## [1] "Kolejny wielki kot" "Kolejny wielki kot"
## [4] "Kolejny wielki kot" "Kolejny wielki kot"
## [7] "Kolejny wielki kot" "Kolejny szybki ptak"
## [10] "Kolejny szybki ptak" "Kolejny szybki ptak"
## [13] "Kolejny szybki ptak"
```

---

## Pętle i instrukcje warunkowe

Instrukcje warunkowe są często wykorzystywane wewnątrz pętli. W sytuacji gdy chcemy powtórzyć pewną operację kilkakrotnie, ale przebieg danej operacji może mieć kilka wariantów / scenariuszy.

Zilustrujemy tę sytuację w poniższym przykładzie.

W zależności od tego czy określony gatunek jest kotem czy ptakiem wyświetlimy inną formułę.

```
## pętla wykona się dla każdego wiersza
for (i in 1:nrow(koty_ptaki)) {
  # wewnątrz pętli wykonanie zależy od warunku
  if (koty_ptaki[i,"druzyna"] == "Kot") {
    cat(koty_ptaki[i,"gatunek"], "to wielki kot.\n")
  } else {
    cat(koty_ptaki[i,"gatunek"], "to ptak.\n")
  }
}
```

```
## Tygrys to wielki i szybki kot.
## Lew to wielki i szybki kot.
```

```
## Jaguar to wielki i szybki kot.  
## Puma to wielki i szybki kot.  
## Leopard to wielki i szybki kot.  
## Gepard to wielki i szybki kot.  
## Irbis to wielki i szybki kot.  
## Jerzyk to ptak.  
## Strus to ptak.  
## Orzel przedni to ptak.  
## Sokol wedrowny to ptak.  
## Sokol norweski to ptak.  
## Albatros to ptak.
```

---

## Pętle i instrukcje warunkowe

Innym częstym zastosowaniem instrukcji warunkowych jest zmiana wartości zmiennej na inną wartość w zależności od tego jaki wystąpił warunek.

W przykładzie poniżej, w pętli wykonujemy instrukcję warunkową, która w jednej sytuacji zmienia wartość zmiennej `liczbaKotów`, a w innej zmiennej `liczbaPtaków`.

```
## inicjujemy zmienne  
liczbaKotow <- 0  
liczbaPtakow <- 0  
## w pętli będziemy zmieniać wartość zmiennych  
## czy kolejny wiersz opisuje koty czy ptaki  
for (i in 1:nrow(koty_ptaki)) {  
  if (koty_ptaki[i,"druzyna"] == "Kot") {
```

```
    liczbaKotow <- liczbaKotow + 1
  } else {
    liczbaPtakow <- liczbaPtakow + 1
  }
}
## zliczamy ile było kotów i ptaków
liczbaKotow

## [1] 7

liczbaPtakow

## [1] 6
```

---

## Podsumowanie instrukcji R

W tym odcinku omawialiśmy instrukcje warunkowe.

Poniżej znajduje się zestawienie wszystkich wykorzystanych w tym odcinku instrukcji.

```
## Instrukcja warunkowa, której wykonanie zależy od losowości
liczbaLosowa <- runif(n = 1)
if (liczbaLosowa < 0.5) {
  cat("Orzeł")
} else {
  cat("Reszka")
}

## Instrukcja warunkowa z samym blokiem if
i <- 9
if (koty_ptaki[i,"druzyna"] == "Kot") {
```

```

cat(koty_ptaki[i,"gatunek"], "to duży kot.")
}

## Instrukcja warunkowa z blokiem if-else
if (koty_ptaki[i,"druzyna"] == "Kot") {
  cat(koty_ptaki[i,"gatunek"], "to duży kot.")
} else {
  cat(koty_ptaki[i,"gatunek"], "to ptak.")
}

## Warunki logiczne można wykonywać na wektorach
koty_ptaki[, "druzyna"] == "Kot"

## Instrukcja warunkowa ifelse() działa też na
ifelse(koty_ptaki[, "druzyna"] == "Kot",
       "Kolejny wielki kot",
       "Kolejny szybki ptak")

```

---

## Podsumowanie instrukcji R

W tym odcinku omawialiśmy instrukcje warunkowe.

Poniżej znajduje się zestawienie wszystkich wykorzystanych w tym odcinku instrukcji.

```

## W pętlach, gdy dla różnych wierszy należy wykonać
## z instrukcji warunkowych
for (i in 1:nrow(koty_ptaki)) {
  # wewnątrz pętli wykonanie zależy od warunku
  if (koty_ptaki[i,"druzyna"] == "Kot") {
    cat(koty_ptaki[i,"gatunek"], "to wielki kot.")
  } else {
    cat(koty_ptaki[i,"gatunek"], "to ptak.")
  }
}

```

```

cat(koty_ptaki[i,"gatunek"], "to ptak.\n")
}
}

## W instrukcjach warunkowych nie tylko możemy
##     ale również zmieniać wartości zmiennych.
## W poniższym przykładzie w zależności od tego
##     inaczej zmieniamy wartości liczników
liczbaKotow <- 0
liczbaPtakow <- 0
for (i in 1:nrow(koty_ptaki)) {
  if (koty_ptaki[i,"druzyna"] == "Kot") {
    liczbaKotow <- liczbaKotow + 1
  } else {
    liczbaPtakow <- liczbaPtakow + 1
  }
}
}
## zliczamy ile było kotów i ptaków
liczbaKotow
liczbaPtakow

```

---

## Zadania

- Napisz instrukcję warunkową, która dla zwierząt lżejszych niż 1kg wypisze `lekkie`, a dla cięższych niż 1 kg `ciężkie`.
- Napisz instrukcję warunkową, która dla zwierząt lżejszych niż 1kg wypisze `lekkie`, cięższych niż 100kg wypisze `ciężkie` a w przedziale 1-100kg wypisze `średnie`. Taki efekt można uzyskać stosując

dwie instrukcje `if()` lub korzystając z funkcji `switch()` (jak działa funkcja `switch()`? To już należy wyczytać z dokumentacji).

- Napisz pętlę i instrukcję warunkową sumującą łączne masy wszystkich ptaków i kotów osobno.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Funkcje

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 10*

*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Funkcje](#)
- [Funkcje](#)
- [Funkcje](#)
- [Funkcje z argumentami](#)
- [Funkcje - argumenty domyślne](#)
- [Funkcje - obsługa różnych scenariuszy](#)
- [Funkcje - obsługa różnych scenariuszy](#)
- [Funkcje - więcej argumentów](#)
- [Funkcje - funkcja w funkcji](#)
- [Funkcje zgłaszające błędy](#)
- [Parametry przekazywane dalej](#)
- [Funkcje zwracające wartości](#)
- [Podsumowanie instrukcji R](#)
- [Podsumowanie instrukcji R](#)
- [Podsumowanie instrukcji R](#)
- [Zadania](#)

# O czym jest ten odcinek

Jedną z ważniejszych cech języków programowania jest możliwość wykorzystywania rozwiązań, które my lub ktoś inny zbudował w przeszłości. Jest to możliwe jeżeli takie dobre rozwiązania zamienimy w funkcje, czyli bloki kodu które łatwo ponownie wykorzystać.

W tym odcinku nauczymy się:

- jak wywoływać funkcje,
- jak tworzyć własne funkcje,
- jak wywoływać funkcje z funkcji.

Do ilustracji wykorzystamy zbiór danych to `koty_ptaki` dostępny w pakiecie `PogromcyDanych`.

```
library(PogromcyDanych)
head(koty_ptaki, 3)
```

##		gatunek	waga	dlugosc	predkosc	habitat
##	Tygrys	Tygrys	300	2.5	60	Azja
##	Lew	Lew	200	2.0	80	Afryka
##	Jaguar	Jaguar	100	1.7	90	Ameryka

---

## Funkcje

Jest wiele powodów, dla których warto korzystać z



funkcji. Trzy najistotniejsze to.

- Funkcje pozwalają na łatwe, ponowne użycie poprzednio opracowanych fragmentów kodu. Dzięki temu możemy wykorzystywać rozwiązania, które już raz zbudowaliśmy i szybciej tworzyć kolejne. Możemy wykorzystywać rozwiązania / funkcje innych osób i dzielić się z nimi własnymi funkcjami.
  - Funkcje pozwalają na logiczny podział programu, na fragmenty, które łatwiej opisać i udokumentować. Łatwiej zrozumieć co się dzieje w programie, kiedy można poznawać poszczególne fragmenty niezależnie.
  - Funkcje pozwalają skrócić długość programu. Podobne bloki kodu można zastąpić wywołaniem funkcji, przez co będzie mniej powtarzających się podobnych bloków. A im krótszy program, tym łatwiej go napisać, łatwiej go zrozumieć i łatwiej znaleźć ewentualne błędy.
- 

## Funkcje

Przywołajmy kod programu z odcinka 8, który rysował prędkość zwierząt za pomocą liczby kropek.

Gdybyśmy chcieli ponownie wykorzystać ten sposób prezentacji danych, musielibyśmy przepisać ten kod na nowo. W tym odcinku zobaczymy jak zamienić użyteczny fragment kodu w jedną lub więcej funkcji, tak by łatwiej można było go używać.

```
## zamieniamy na zmienną napisową, by była dobre
koty_ptaki$gatunek <- as.character(koty_ptaki$gatunek)

## Dla każdego wiersza w tabeli `koty_ptaki` wyliczamy
for (i in 1:nrow(koty_ptaki)) {
  n_kropek <- koty_ptaki[i,"predkosc"] / 5
  for (j in 1:n_kropek) {
    cat(".")
  }
  cat(" ", koty_ptaki[i,"gatunek"], "\n")
}
```

```
## ..... Tygrys
## ..... Lew
## ..... Jaguar
## ..... Puma
## ..... Leopard
## ..... Gepard
## ..... Irbis
## ..... Jerzyk
## ..... Strus
## ..... Orzeł przelotny
## ..... Sokół wędrowny
## ..... Sokół norweski
## ..... Albatros
```



# Funkcje

Do tworzenia funkcji służy słowo `function`. Dla każdej funkcji należy określić listę jej argumentów oraz tak zwane ciało funkcji, czyli listę instrukcji, które funkcja wykonuje.

```
function(argumenty_rozdzielone_przecinkiem) {  
  wyrażenie  
}
```

gdzie `argumenty_rozdzielone_przecinkiem` to lista rozdzielonych przecinkiem argumentów (może być też pusta lub jednoelementowa), a `wyrażenie` to instrukcje, które mają być wykonane w ramach wywołania funkcji.

Wynikiem powyższej instrukcji jest funkcja, którą należy przypisać do zmiennej, by móc jej później używać. Dlatego najczęstsze deklaracje mają również przypisanie funkcji do zmiennej, tak jak w deklaracji poniżej.

```
nazwa_funkcji <- function(argumenty_rozdzielone  
  wyrażenie  
)
```

Warto jednak pamiętać, że nazwa funkcji nie jest obowiązkowa i - jak się okaże - często będziemy używać funkcji anonimowych, czyli funkcji bez nazwy.

---

# Funkcje z argumentami

Dla naszego przykładu z rysowaniem kropek stworzymy funkcję, którą przypiszemy do zmiennej `rysuj_kropki`. Funkcja przyjmować będzie jeden argument `n_kropek` i narysuje na ekranie dokładnie tyle kropek ile jej każemy.

Do rysowania kropek wykorzystamy pętlę `for`.

```
rysuj_kropki <- function(n_kropek) {
  # w pętli rysujemy n_kropek
  for (j in 1:n_kropek) {
    cat(".")
  }
}
```

Przykładowe wywołanie tej funkcji, rysujące 20 kropek, wygląda następująco.

```
rysuj kropki (n kropek = 20)
```

## #

Jeżeli podajemy argumenty zgodnie z domyślną kolejnością, to nie musimy wskazywać ich nazw. Dotyczy to zarówno funkcji, które sami tworzymy, jak i *fabrycznych* funkcji.

W poniższym przykładzie podajemy wartość dla argumentu, a ponieważ nie ma nazwy argumentu, wartość

ta przypisana będzie do pierwszego (w tym przypadku jednego) argumentu.

```
rysuj_kropki(20)
```

```
## .....
```

---

## Funkcje - argumenty domyślne

Jeżeli funkcja jest często wywoływana z tą samą wartością parametru, to aby oszczędzić sobie pisania, warto wskazać tę wartość jako wartość domyślną.

Deklarując funkcję, w liście argumentów wpisujemy wartość domyślną po znaku `=`. Poniższy przykład tworzy nową funkcję (nadpisując przy okazji poprzednią deklarację) z domyślną wartością parametru `n_kropek`.

```
rysuj_kropki <- function(n_kropek = 20) {  
  for (j in 1:n_kropek) {  
    cat(". ")  
  }  
}
```

Dzięki wartościom domyślnym, jeżeli nie podamy wartości argumentu, to zostanie on zastąpiony domyślną wartością. Jest to bardzo wygodne w sytuacji, gdy funkcja ma wiele argumentów.

```
rysuj_kropki()
```

```
## .....
```

Jeżeli podana zostanie wartość argumentu, to nadpisze ona domyślną deklarację.

```
rysuj_kropki(35)
```

```
## .....
```

---

## Funkcje - obsługa różnych scenariuszy

Chcielibyśmy, by funkcje działały tak jak to zaplanowaliśmy. Rzeczywistość jest jednak taka, że funkcje często zawierają błędy lub sytuacje, o których nie pomyśleliśmy.

Przykładowo, uruchamiając funkcję `rysuj_kropki()` z argumentem 0 spodziewalibyśmy się 0 kropek, a tym czasem...

```
rysuj_kropki(0)
```

```
## ..
```

Dlaczego tak jest?

.

Otóż dlatego, że w pętli `1:n_kropek`, która znajduje się w deklaracji funkcji wektor `1:0` jest w rzeczywistości dwuelementowy.

```
1:0
```

```
## [1] 1 0
```

Jeżeli więc chcemy by dla `n_kropek = 0` funkcja działała poprawnie musimy inaczej obsłużyć tę sytuację.

Jak zaradzić temu problemowi?

---

## Funkcje - obsługa różnych scenariuszy

Ten problem można obsłużyć na kilka sposobów.  
Przedstawimy taki, który wymaga użycia instrukcji `if()`.

Sprawdzimy, czy `n_kropek` jest równe lub mniejsze niż zero (jeżeli tak to nic nie rysuj), czy jest większe od zera (rysuj kropki).

```
rysuj_kropki <- function(n_kropek = 20) {  
  # czy liczba kropek do wyświetlenia jest więł  
  if (n_kropek > 0) {  
    for (j in 1:n_kropek) {  
      cat(".")  
    }  
  }  
}
```

```
}
```

Działa dla argumentu równego 0.

```
rysuj_kropki(0)
```

Działa dla domyślnych argumentów.

```
rysuj_kropki()
```

```
## .....
```

Działa dla innych wartości.

```
rysuj_kropki(30)
```

```
## .....
```

---

## Funkcje - więcej argumentów

Funkcje mogą mieć więcej argumentów i każdy z nich może mieć (lub nie) wartość domyślną.

Jeżeli wywołując funkcję podajemy argumenty w kolejności innej niż domyślna, musimy podać nazwę argumentu, który określamy.

Aby to zilustrować dodajmy argument opisujący znak, który ma być rysowany.



```
rysuj_kropki <- function(n_kropek = 20, znak =  
  if (n_kropek > 0) {  
    for (j in 1:n_kropek) {  
      cat(znak)  
    }  
  }  
}
```

I seria wywołań. Określamy drugi argument, pierwszy pozostaje domyślny, musimy wskazać nazwę.

```
rysuj_kropki(znak="X")  
  
## XXXXXXXXXXXXXXXXXXXXXXXX
```

Określamy pierwszy argument, drugi pozostanie domyślny, nie musimy podawać nazwy.

```
rysuj_kropki(35)  
  
## .....
```

Możemy argumenty podawać w dowolnej kolejności, ale musimy podawać je z nazwami.

```
rysuj_kropki(znak="X", n_kropek = 30)  
  
## XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

---

## Funkcje - funkcja w funkcji

Tworząc nowe funkcje możemy używać w nich innych funkcji, także tych które wcześniej zdefiniowaliśmy.

Przykładowo, poniżej zbudujemy funkcję rysującą wykres kropkowy dla wektora napisów i wektora wartości.

```
rysuj_wykres_kropkowy<- function(nazwy, wartosci) {  
  # zakładamy, że oba wektory są tej samej długości  
  # funkcja seq_along(nazwy) tworzy sekwencję indeksów  
  for (i in seq_along(nazwy)) {  
    rysuj_kropki(wartosci[i])  
    cat(" ", nazwy[i], "\n")  
  }  
}
```

I przykładowe wywołanie.

```
rysuj_wykres_kropkowy(koty_ptaki$gatunek, koty_ptaki$waga)  
  
## ..... Tygrys  
## ..... Lew  
## ..... Jaguar  
## ..... Puma  
## ..... Leopard  
## ..... Gepard  
## ..... Irbis  
## ..... Jerzyk  
## ..... Strus  
## ..... Orzeł  
## ..... Sokół wędrowny  
## ..... Sokół norweski  
## ..... Albatros
```

Warto utrzymywać każdą z funkcji o niewielkiej długości, aby łatwo było opisać i zapamiętać co ta funkcja robi. Z tego też powodu warto funkcje dobrze nazywać i dokumentować.

---

## Funkcje zgłaszające błędy

W funkcji z poprzedniego slajdu założyliśmy, że oba wektory są równej długości.

A co jeżeli nie są? Może użytkownik nie wie, że powinny być?

Takie założenia warto sprawdzać, np. używając instrukcji `if()`. Jeżeli założenie jest ważne i bez jego spełnienia nie można kontynuować działania funkcji, to możemy przerwać działanie funkcji i zwrócić błąd poleceniem `stop()`.

```
rysuj_wykres_kropkowy <- function(nazwy, wartosci) {  
  # czy oba argumenty mają równą długość?  
  if (length(nazwy) != length(wartosci)) {  
    # funkcja stop() przerywa działanie funkcji  
    stop("Argumenty mają różną długość! ", length(nazwy))  
  }  
  # jeżeli wszystkie warunki są spełnione to możemy kontynuować  
  for (i in seq_along(nazwy)) {  
    rysuj_kropki(wartosci[i])  
  }  
}
```

```
cat(" ", nazwy[i], "\n")
}  
}
```

Co się stanie, jeżeli wywołamy tę funkcję z niepoprawnymi argumentami.

```
rysuj_wykres_kropkowy(koty_ptaki$gatunek, 5)
```

```
Error in rysuj_wykres_kropkowy(koty_ptaki$gatunek, 5) :  
Argumenty mają różną długość! 13 oraz 1
```

---

## Parametry przekazywane dalej

W liście argumentów funkcji dozwolony jest też specjalny argument `...`. Używając go można przekazać wszystkie pozostałe argumenty dalej, do funkcji wewnętrznych.

W przykładzie poniżej operator `...` pojawia się w dwóch miejscach, w liście argumentów funkcji

`rysuj_wykres_kropkowy()` i w liście argumentów funkcji `rysuj_kropki()`.

Wszystkie argumenty dla funkcji

`rysuj_wykres_kropkowy()`, których nazwa jest różna od nazwy i wartości będą przekazane dalej do funkcji `rysuj_kropki()`.

```
rysuj_wykres_kropkowy <- function(nazwy, wartosci, ...)
```

```
for (i in seq_along(nazwy)) {  
  rysuj_kropki(wartosci[i], ...)  
  cat(" ", nazwy[i], "\n")  
}  
}
```

Zilustrujmy to na przykładzie. Wywołujemy `rysuj_wykres_kropkowy()` z trzema argumentami, trzeci, czyli `znak = "x"` zostanie przekazany do `rysuj_kropki()` dzięki czemu na wykresie pojawią się znaki x.

```
rysuj_wykres_kropkowy(nazwy = LETTERS[1:5], wa:
```

```
## X   A  
## XX  B  
## XXX C  
## XXXX D  
## XXXXX E
```

---

## Funkcje zwracające wartości

Funkcje zawdzięczają swoją nazwę temu, że zwracają wartości.

Ale te funkcje, które powyżej stworzyliśmy wypisywały wyniki na ekranie.

Domyślnie wynikiem funkcji jest wynik ostatniego

wyrażenia w funkcji. Innym sposobem na wskazanie wyniku jest użycie funkcji `return()`, która przerywa działanie funkcji i jako wynik zwraca wartość argumentu funkcji `return()`.

Zilustrujmy to na poniższym przykładzie. Funkcja `suma_n_liczb_losowych()` losuje `n` liczb i jako wynik zwraca ich sumę.

Jeżeli jednak jako argument podana jest wartość mniejsza niż 1, to działanie funkcji jest przerywane i jako wynik zwracana jest wartość 0.

```
suma_n_liczb_losowych <- function(n = 10) {  
  if (n < 1) {  
    return(0)  
  }  
  sum(runif(n))  
}
```

Wywołajmy tę funkcję. Jeżeli jej wynik nie zostanie przypisany do żadnej zmiennej to zostanie wyświetlony na ekranie.

```
suma_n_liczb_losowych(10)
```

```
## [1] 3.78565
```

I wynik, gdy argument jest mniejszy niż 1.

```
suma_n_liczb_losowych(-1)
```

---

## Podsumowanie instrukcji R

W tym odcinku omawialiśmy funkcje, z argumentami, bez argumentów, z wynikami i bez wyników.

Poniżej znajduje się zestawienie wszystkich wykorzystanych w tym odcinku instrukcji.

```
## Tworzymy funkcję wyświetlającą n_kropek kropki
rysuj_kropki <- function(n_kropek) {
  for (j in 1:n_kropek) {
    cat(".")
  }
}

## Funkcje można wywołać podając nazwę argumentu
rysuj_kropki(n_kropek = 20)
rysuj_kropki(20)

## Tworzymy funkcję z domyślną wartością argumentu
rysuj_kropki <- function(n_kropek = 20) {
  for (j in 1:n_kropek) {
    cat(".")
  }
}

## Dodajemy obsługę argumentów mniejszych niż 0
rysuj_kropki <- function(n_kropek = 20) {
  if (n_kropek > 0) {
```

```
for (j in 1:n_kropek) {  
  cat(".")  
}  
}  
}
```

---

## Podsumowanie instrukcji R

Poniżej znajduje się zestawienie wszystkich wykorzystanych w tym odcinku instrukcji.

```
## Dodajemy drugi argument, określający jakie :  
rysuj_kropki <- function(n_kropek = 20, znak =  
  if (n_kropek > 0) {  
    for (j in 1:n_kropek) {  
      cat(znak)  
    }  
  }  
}
```

```
## Możemy podawać argumenty w dowolnej kolejnos  
rysuj_kropki(znak="X", n_kropek = 30)
```

```
## Wewnątrz jednej funkcji wywołujemy inną  
rysuj_wykres_kropkowy<- function(nazwy, wartosc  
  for (i in seq_along(nazwy)) {  
    rysuj_kropki(wartosci[i])  
    cat(" ", nazwy[i], "\n")  
  }  
}
```

```
## W przypadku gdy argumenty są złe, zatrzymuje
```



```
rysuj_wykres_kropkowy <- function(nazwy, wartos
# czy oba argumenty mają równą długość?
if (length(nazwy) != length(wartosci)) {
  stop("Argumenty mają różną długość! ", leng
}
for (i in seq_along(nazwy)) {
  rysuj_kropki(wartosci[i])
  cat(" ", nazwy[i], "\n")
}
}
```

---

## Podsumowanie instrukcji R

Poniżej znajduje się zestawienie wszystkich wykorzystanych w tym odcinku instrukcji.

```
## Argumenty możemy przekazywać dalej używając
rysuj_wykres_kropkowy <- function(nazwy, wartos
  for (i in seq_along(nazwy)) {
    rysuj_kropki(wartosci[i], ...)
    cat(" ", nazwy[i], "\n")
  }
}

## Przykładowe wywołanie, ostatni argument zost
rysuj_wykres_kropkowy(nazwy = LETTERS[1:5], wa

## Funkcja która zwraca wynik liczbowy
suma_n_liczb_losowych <- function(n = 10) {
  if (n < 1) {
    return(0)
  }
}
```

```
sum(runif(n))  
}  
  
## Dwa przykładowe wywołania, zwracają wynik r  
suma_n_liczb_losowych(10)  
## poprzez wywołanie funkcji return()  
suma_n_liczb_losowych(-1)
```

---

## Zadania

- Napisz funkcję, która otrzymuje argument liczbowy, a następnie wypisuje na ekran wartości od argumentu do jeden.
- Napisz funkcję, przyjmuje argument liczbowy `n`, a następnie rysuje kwadrat o boku `n` wypełniony znakami `x`.
- Napisz funkcję, przyjmuje argument liczbowy `n`, a następnie rysuje kwadrat o boku `n` ze znakami `x` na brzegu i pusty w środku.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Cechy ilościowe

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 11*

*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Co to znaczy: cecha ilościowa](#)
- [Wczytanie danych](#)
- [Statystyki opisowe](#)
- [Wczytanie danych](#)
- [Statystyki opisowe](#)
- [Statystyki opisowe](#)
- [Statystyki opisowe](#)
- [Statystyki opisowe](#)
- [Statystyki opisowe](#)
- [Zadania](#)
- [Brakujące wartości](#)
- [Brakujące wartości](#)
- [Zadania:](#)
- [Graficzne statystyki opisowe - wykres słupkowy](#)
- [Graficzne statystyki opisowe - wykres słupkowy](#)
- [Graficzne statystyki opisowe - wykres pudełkowy](#)
- [Graficzne statystyki opisowe - wykres pudełkowy](#)

- [Graficzne statystyki opisowe - histogram](#)
- [Graficzne statystyki opisowe - histogram](#)
- [Graficzne statystyki opisowe - wykres punktowy](#)
- [Graficzne statystyki opisowe - wykres punktowy](#)
- [Zadania:](#)

## O czym jest ten odcinek

Analizując dane spotkamy się z różnymi rodzajami zmiennych. W klasycznej statystyce najbardziej popularne są zmienne ilościowe, którym poświęcony jest ten odcinek.

W tym odcinku nauczymy się:

- jakie zmienne / cechy określa się terminem *cechy ilościowe*,
- jakie podstawowe operacje można wykonywać na cechach ilościowych,
- jak podsumowywać / opisywać cechy ilościowe.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012`, oba dostępne w pakiecie `PogromcyDanych`.

---

# Co to znaczy: cecha ilościowa

Najbliższe pięć odcinków przedstawia pięć podstawowych typów danych: ilościowe, jakościowe, napisy, wartości logiczne i daty. Nie wyczerpują one jeszcze listy wszystkich możliwych typów, można by wyróżnić bardziej specjalistyczne typy (np. współrzędne geograficzne), ale te pięć występuje w 90% analiz, więc na nich się skupimy.

Słowo ‘typ’ ma w tym przypadku to samo znaczenie co słowo rodzaj, możemy więc mówić o typach danych jak i o rodzajach danych. W analizie danych częściej stosowane jest słowo „typ”, więc będziemy się nim posługiwać.

Cechy ilościowe to takie, które opisują ilości - wysokość, długość, wagę, prędkość, powierzchnię, wiek itp. Często stosuje się też określenie cechy liczbowe ponieważ są one wyrażane za pomocą liczb. Jednak nie zawsze to co jest opisane liczbą jest cechą ilościową. Przykładowo PESEL lub kod pocztowy opisać można liczbą ale nie reprezentują one ilości czy wielkości.

Cechy ilościowe mogą być fizyczne (masa, długość, prędkość) lub nie (iloraz inteligencji, wielkość długu publicznego, procentowy poziom zgodności z poglądami

jakiejś partii).

Mogą przyjmować bardzo wiele różnych wartości (np. wzrost w milimetrach) lub tylko kilka (np. liczba rąk).

Ponieważ jednak są opisane za pomocą liczb, można na nich wykonywać kilka wspólnych operacji. Przyjrzyjmy się im.

---

## Wczytanie danych

Dane od których rozpoczniemy przykłady to `koty_ptaki` z pakietu `PogromcyDanych`. Aby te dane wczytać, wystarczy włączyć pakiet, instrukcja jak to zrobić znajduje się w odcinku 2.

Włączmy pakiet i użyjmy funkcji `head()` by wyświetlić pierwsze sześć wierszy. Które z tych zmiennych to zmienne ilościowe?

```
library(PogromcyDanych)
head(koty_ptaki)
```

##		gatunek	waga	dlugosc	predkosc	habita
##	Tygrys	Tygrys	300	2.5	60	Azj
##	Lew	Lew	200	2.0	80	Afryl
##	Jaguar	Jaguar	100	1.7	90	Ameryl
##	Puma	Puma	80	1.7	70	Ameryl

##	Leopard	Leopard	70	1.4	85	Az:
##	Gepard	Gepard	60	1.4	115	Afryl

Każdy wiersz opisuje jeden gatunek. Zmienne ilościowe opisane są liczbami. W tym przypadku nie ma niespodzianek - zmiennymi ilościowymi są: waga, długość, prędkość i żywotność.

---

## Statystyki opisowe

Zbiór danych `koty_ptaki` składa się z 13 wierszy, więc każda zmienna ilościowa to 13 liczb.

Aby coś powiedzieć o każdej ze zmiennych wygodnie jest scharakteryzować je jedną lub kilkoma wskaźnikami, takimi jak średnia czy mediana (mediana to wartość która pojawi się w środku gdyby te 13 liczb posortować).

Jak sprawdzić ile wynosi średnia lub mediana dla wagi? Aby policzyć średnią możemy wykorzystać funkcję `mean()`, a do wyznaczenia mediany funkcję `median()`.

```
mean(koty_ptaki$waga)
```

```
## [1] 78.59615
```

```
median(koty_ptaki$waga)
```

```
## [1] 60
```

```
sort(koty_ptaki$waga)
```

```
##      [1]      0.05      0.70      2.00      4.00      5.00      50  
##     [11]    150.00    200.00    300.00
```

W ostatnim wierszu wyświetliliśmy wszystkie wagi po ich posortowaniu. Funkcja `sort()` domyślnie sortuje liczby rosnąco. Możemy zatem przy okazji sprawdzić, że rzeczywiście środkową wartością jest 60.

Przy okazji warto zauważyć że średnia i mediana znacznie się różnią - o prawie 20 kilogramów. Jest to wynikiem tego, że najmniejsze i największe wartości różnią się od siebie o ponad dwa rzędy wielkości. W takich sytuacjach średnia często jest odległa od mediany.

---

## Wczytanie danych

Zbiór danych `koty_ptaki` składa się z 13 wierszy. Można cały ten zbiór danych wyświetlić na ekranie. Nie zawsze potrzebujemy więc specjalnych statystyk opisowych by zrozumieć co się dzieje w takich małych zbiorach danych.

Dlatego, dalsze ćwiczenia ze zmiennymi ilościowymi przeprowadzimy na znacznie większym zbiorze danych z ponad 200 tysiącami wartości, o nazwie `auta2012`, który znajduje się również w pakiecie `PogromcyDanych`.



Opis tego zbioru danych znaleźć można w odcinku [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

Wczytajmy ten zbiór danych i przyjrzyjmy się trzem pierwszym wierszom.

```
library(PogromcyDanych)
head(auta2012, 3)

## Source: local data frame [3 x 28]
##
##      Cena Waluta Cena.w.PLN Brutto.netto KM
## 1  49900     PLN      49900      brutto 140
## 2  88000     PLN      88000      brutto 156
## 3  86000     PLN      86000      brutto 150
## Variables not shown: Liczba.drzwi (fctr), Przebieg.w.km (dbl), Rodzaj.paliwa (fctr),
##      (fctr), Kraj.aktualnej.rejestracji (fctr), Pojazd.uszkodzony (fctr), Skrzynia.biegow
##      Status.pojazdu.sprowadzonego (fctr), Wypos
##      Rodzaj.paliwa.posortowany (fctr), Kolor_na
##      Wyposazenie.dodatkowe_napis (chr), czy_me
##      (lgl), szyby (lgl), MarkaModel (chr)
```

Które z tych zmiennych to zmienne ilościowe?

---

## Statystyki opisowe

Przyjrzyjmy się takim cechom jak *Cena.w.PLN* lub *Przebieg.w.km*. Cechy te są opisane przez liczby oraz

przedstawiają ilości.

Pierwsza z nich przedstawia *ilość* pieniędzy, za którą sprzedawca chce sprzedać samochód, druga opisuje *ilość* kilometrów, którą przejechał wystawiony na aukcji samochód.

Na zmiennych ilościowych wykonać możemy kilka operacji. Używając funkcji `mean()` możemy policzyć średnią cenę wystawionych aut, a funkcją `median()` policzymy medianę, podobnie jak dla poprzedniego zbioru danych.

```
mean(auta2012$Cena.w.PLN)
```

```
## [1] 35755.11
```

```
median(auta2012$Cena.w.PLN)
```

```
## [1] 19900
```

Średnia cena aut to ponad 35 tysięcy, ale ponad połowa aut jest tańsza niż 20 tysięcy.

Jak to możliwe? Widoczne jest kilka bardzo drogich aut, które przesuwają średnią do góry.

---

## Statystyki opisowe

Wczytując dane ze zbioru, którego sami nie tworzyliśmy, często chcemy zobaczyć skrajne wartości dla poszczególnych zmiennych. Skrajne czyli najmniejsze i największe.

Używając funkcji `min()` i `max()` możemy wyznaczyć minimalną i maksymalną cenę w tym zbiorze danych.

```
min(auta2012$Cena.w.PLN)
```

```
## [1] 400
```

```
max(auta2012$Cena.w.PLN)
```

```
## [1] 11111111
```

Najniższa cena to 400 złotych, w co jestem w stanie uwierzyć tym bardziej, że część tych cen dotyczy aut uszkodzonych.

Ale najwyższa to już 11 milionów 111 tysięcy 111. Być może ktoś wpisał taką cenę, ale to chyba bardziej żart niż rzeczywista wartość auta.

Obie te wartości, minimum i maksimum, jako dwuelementowy wektor zwraca funkcja `range()`.

```
range(auta2012$Cena.w.PLN)
```

```
## [1] 400 11111111
```

---

# Statystyki opisowe

Używając funkcji `summary()` możemy wyznaczyć sześć podstawowych charakterystyk, czyli: minimum, 1. kwartyl, medianę (2. kwartyl), średnią, 3. kwartyl i maksimum.

Pięć z tych charakterystyk, tj. minimum, 1., 2., 3. kwartyl i maksimum to tak zwana piątka Tukeya, a więc pięć liczb, które dzielą wartości na cztery równoliczne przedziały.

- Jedna czwarta samochodów ma cenę niższą od 1. kwartyla.
- Jedna czwarta samochodów ma cenę wyższą od 1. kwartyla ale niższą od mediany (2. kwartyla).
- Jedna czwarta samochodów ma cenę wyższą od mediany (2. kwartyla) ale niższą od 3. kwartyla.
- Jedna czwarta samochodów ma cenę wyższą od 3. kwartyla.

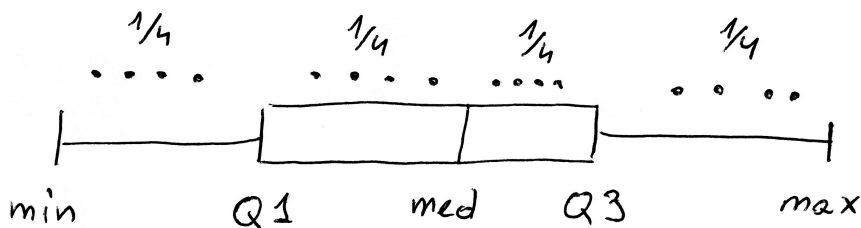
Opis cechy ilościowej za pomocą tych sześciu liczb bardzo wiele o danej cesze mówi. Przyjrzyjmy się tym charakterystykom na przykładzie cen aut

```
summary(auta2012$Cena.w.PLN)
```

##	Min.	1st Qu.	Median	Mean	3rd Qu
##	400	10900	19900	35760	37470

Te pięć liczb, dzielących wszystkie obserwacje na cztery

równoliczne części ilustruje wykres pudełkowy.



## Statystyki opisowe

Opiszmy wyniki funkcji `summary()` na przykładzie

```
summary(auta2012$Cena.w.PLN)
```

##	Min.	1st Qu.	Median	Mean	3rd Qu
##	400	10900	19900	35760	37470

- Minimalna oferta opiewa na kwotę na 400 pln.
- Jedna czwarta ogłoszeń oferuje samochód w cenie niższej niż 10 900 (a ponieważ wszystkich aut jest ponad 200 tysięcy więc szybko możemy policzyć, że ponad 50 tysięcy ofert znajduje się w tym przedziale).
- Połowa ogłoszeń oferuje samochód w cenie niższej niż 19 900.
- Trzy czwarte ogłoszeń oferuje samochód w cenie niższej niż 37 470, tym samym jedna czwarta ofert jest na kwotę wyższą.

- Maksymalna cena ofertowa to wspomniane ponad 11 milionów.
- Również i w tym przykładzie średnia cena (35 760) jest znacznie - prawie dwukrotnie - wyższa od ceny środkowej. Dzieje się tak dlatego, że większość aut oferowanych jest w niższych cenach, ale niewielka część bardzo drogiech aut znacznie podnosi wartość średniej. Jest to jeden z argumentów przeciwko używaniu średniej, która jest podatna na skrajne obserwacje.

Opisując zmienną ilościową przydatna może być funkcja `quantile()`, która wyznacza kwantyle określonego rzędu ze zbioru danych. Kwantyle rzędu  $x$  to taka liczba, że  $100 \cdot x\%$  wartości jest mniejszych niż ten kwantyl.

Niezbyt jasne? Wtedy może na przykładzie.

```
quantile(auta2012$Cena.w.PLN, c(0.01, 0.1, 0.25, 0.5, 0.75, 0.9))
```

##	1%	10%	25%	50%	75%	90%
##	1500.0	5500.0	10900.0	19900.0	37470.0	72500.0

Kwantyl rzędu 1% to 1500, czyli co setna oferta jest tańsza niż 1500 PLN. Kwantyle rzędu 10% to 5500, czyli jedna dziesiąta ofert jest tańsza niż 5.5 tys. Kwantyl rzędu 90% to 72.5 tys, czyli co dziesiąty samochód jest droższy niż 72 500 pln.

# Statystyki opisowe

Dotąd omówiliśmy statystyki opisowe dla jednej zmiennej. Czasem interesuje nas zależność dwóch zmiennych.

Najbardziej typową statystyką opisową dla pary zmiennych ilościowych jest korelacja. Można ją wyznaczyć używając funkcji `cor()`. Zobaczmy czy istnieje zależność pomiędzy rokiem produkcji a ceną.

```
cor(auta2012$Rok.produkcji, auta2012$Cena.w.PL)
```

```
## [1] 0.3419877
```

Zależność jest dodatnia, a więc im rok produkcji auta wyższy (auto jest młodsze), tym jest ono droższe. Korelacja przyjmuje wartości od -1 do 1. Wartość 0.34 to przeciętna korelacja, pewnie część z nas mogłaby się spodziewać wyższej. Ale tak się składa, że mamy tutaj wymieszane różne marki i modele. Funkcja `cor()` domyślnie liczy korelację Pearsona, ale może też liczyć inne korelacje, np. Spearmana. Co to dokładnie znaczy i jak z niej korzystać, to omówimy w sezonie 3.

Inną przydatną funkcją, jest możliwość sprawdzenia ile aut (ile wierszy danych) ma określoną cechę większą lub mniejszą od jakiejś wartości.

Na przykład, ile jest ofert sprzedaży aut tańszych niż 5 tysięcy? Lub ile jest ofert sprzedaży aut droższych niż milion?

```
sum(auta2012$Cena.w.PLN < 5000)
```

```
## [1] 18559
```

```
sum(auta2012$Cena.w.PLN > 1000000)
```

```
## [1] 44
```

Każda oferta, która spełnia warunek w nawiasie liczy się jako 1, a która nie spełnia jako 0. Suma tych wartości to suma aut spełniających dany warunek.

Więcej o tym dlaczego i jak działa to sumowanie omówimy w odcinku 13 o cechach logicznych.

---

## Zadania

- W zbiorze danych *auta2012* aż 7 cech to cechy ilościowe. Wymień które.
- Jedną z cech ilościowych jest *Rok.produkcji*. Jaki jest medianowy/półówkowy rok produkcji oferowanych aut? Wszystkie te oferty były złożone w roku 2012, jaki był medianowy/półówkowy wiek



oferowanego auta?

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

---

## Brakujące wartości

Może się tak zdarzyć, że oferta sprzedaży nie zawiera wszystkich informacji o aucie. Np. brakuje informacji o przebiegu. Takie braki danych trzeba jakoś oznaczyć. Nie można w ich miejsce wstawić np. 0, ponieważ czymś innym jest przebieg równy 0, a czymś innym jest brak informacji o przebiegu.

Potrzebna jest więc jakaś specjalna wartość, która będzie oznaczała brakujące wartości. W programie R taką wartością jest `NA` (skrót od *not available*, czyli *niedostępne*).

Jak zauważyć, że w danych występują wartości niedostępne? Ich liczba będzie wymieniona w wyniku funkcji `summary()`. Dla zmiennej `Przebieg.w.km` brakujących wartości jest prawie 40 tysięcy. Czyli dla znacznej części aut nie podano informacji o ich przebiegu.

Pozostałe charakterystyki, takie jak minimum, maksimum i

mediana, są liczone oczywiście na pozostałych 160 tysiącach aut, dla których przebieg został podany.

```
summary(auta2012$Przebieg.w.km)
```

##	Min.	1st Qu.	Median	Mean	3rd Qu.
##	1.000e+00	8.500e+04	1.400e+05	1.472e+05	1.800e+05

## Brakujące wartości

Jakie są konsekwencje występowania brakujących wartości?

Najpoważniejszą jest taka, że części statystyk nie można policzyć.

Przykładowo, średnia z 1 i 3 to 2. Ale ile wynosi średnia z 1 i *nie wiadomo*?

Dlatego gdy liczymy średnie lub inne statystyki z wektorów, które zawierają wartości nieokreślone to wynik też jest wartością nieokreśloną `NA`.

```
mean(auta2012$Przebieg.w.km)
```

```
## [1] NA
```

```
min(auta2012$Przebieg.w.km)
```

```
## [1] NA
```

```
max(auta2012$Przebieg.w.km)
```

```
## [1] NA
```

To logiczne. Jeżeli nie wszystkie wartości są znane, to nie można policzyć ani średniej, ani innej statystyki. Logiczne ale też mało praktyczne. Często chcielibyśmy policzyć średnią, medianę, minimum tylko z tych wartości które są znane.

Aby to zrobić w programie R, do funkcji należy dodać argument `na.rm=TRUE`, gdzie `na.rm` to skrót od *remove NA*, czyli usunąć wartości brakujące.

```
mean(auta2012$Przebieg.w.km, na.rm=TRUE)
```

```
## [1] 147167.4
```

```
min(auta2012$Przebieg.w.km, na.rm=TRUE)
```

```
## [1] 1
```

```
max(auta2012$Przebieg.w.km, na.rm=TRUE)
```

```
## [1] 1e+09
```

---

## Zadania:

- Która cecha jest najmniej kompletna? Dla której

cechy liczba brakujących wartości jest największa?

- Jaka jest mediana/połówkowa wielkość silnika (`Pojemnosc.skokowa`)?

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

---

## Graficzne statystyki opisowe - wykres słupkowy

Powyżej opisane statystyki liczbowe można przedstawić graficznie. Prezentacja graficzna ma tę zaletę, że wyćwiczone oko potrafi szybko wiele dowiedzieć się z danych.

Opisując zmienną ilościową mamy do wyboru różne sposoby prezentacji, w zależności od tego, ile informacji chcemy przedstawić.

Przypuśćmy, że chcemy pokazać wyłącznie średnią (np średni przebieg). Jedną liczbę. Zrobimy to za pomocą wykresu słupkowego. Wykres słupkowy można w programie R wyznaczyć funkcją `barplot()`.

*Zauważmy, że jeżeli wynik przypisania obejmujemy w*

*nawiasy, to wyświetli się on na ekranie*

```
(srednia <- mean(auta2012$Przebieg.w.km, na.rm=
## [1] 147167.4
barplot(srednia)
```



---

## Graficzne statystyki opisowe - wykres słupkowy

Jeden słupek wygląda dziwnie, mało czytelnie. To dlatego,

że wykres przedstawia jedną liczbę, trudno więc do czegoś się na nim odnieść.

Zobaczmy jak wyglądają średnie przebiegi w grupach, na przykład w zależności od rodzaju paliwa.

Do liczenia średniej w grupach wykorzystamy funkcję `tapply()`, jako pierwszy argument przyjmuje cechę ilościową (tutaj zmienną z przebiegiem), jako drugi informacje o grupach (tutaj, rodzaje paliwa) a jako trzeci nazwę funkcji, która ma w każdej grupie być wykonana (i następnie dodatkowe argumenty).

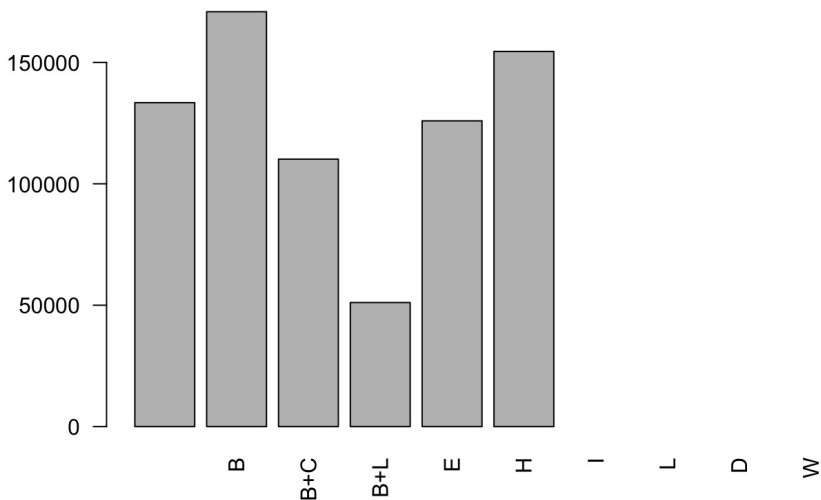
Wynikiem będzie wektor ze średnimi. Możemy teraz użyć funkcji `barplot()`, argument `las=2` powoduje, że etykiety na osi OX pojawiają się w pionie, co ułatwi ich odczytanie.

Więcej informacji o tym jak zrobić dobrze wyglądający wykres słupkowy przedstawimy w sezonie 2.

```
(srednie <- tapply(auta2012$Przebieg.w.km,
                  auta2012$Rodzaj.paliwa,
                  mean, na.rm=TRUE))
```

##		B	B+C	B+L	
##	133414.90	170875.89	110166.93	51097.37	1259
##	L	D	W		
##	NA	NA	NA		

```
barplot(srednie, las=2)
```



---

## Graficzne statystyki opisowe - wykres pudełkowy

Ciekawszy będzie wykres, przedstawiający więcej informacji.

W statystykach opisowych pokazaliśmy funkcję `summary()`, prezentującą sześć charakterystycznych informacji o rozkładzie (min, max, kwartyły i średnią). Pięć z nich, za wyjątkiem średniej, przedstawia wykres

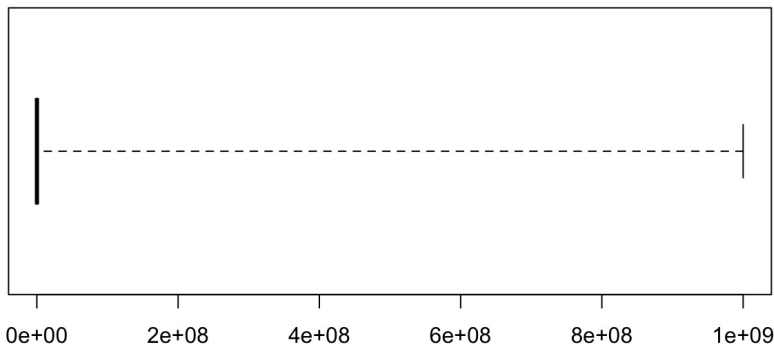
pudełko-wąsy, który w R można wykonać używając funkcji `boxplot()`.

W poniższym przykładzie przedstawiamy te pięć liczb dla zmiennej `Przebieg.w.km`. Argument `horizontal=TRUE` powoduje, że wykres rysowany jest poziomo, argument `range = 0` powoduje, że nie są wyznaczane wartości odstające (a nie mówiliśmy co to za wartości, więc ich nie wyznaczamy - wrócimy do tego tematu później).

```
summary(auta2012$Przebieg.w.km)
```

##	Min.	1st Qu.	Median	Mean	3rd Qu.
##	1.000e+00	8.500e+04	1.400e+05	1.472e+05	1.800e+05

```
boxplot(auta2012$Przebieg.w.km,  
        horizontal=TRUE, range = 0)
```





# Graficzne statystyki opisowe - wykres pudełkowy

Co za dziwny wykres. Co jest nie tak? Spójrzmy na oś na wykresie, rozciąga się do  $10^9 = 1\,000\,000\,000$  km. Patrząc na wynik funkcji `summary()` widzimy że faktycznie, któryś wiersz zawiera tak dużą wartość. Najprawdopodobniej te duże wartości to wynik błędu w danych. Mała jest szansa, że jakiekolwiek auto na świecie miało choćby zbliżony przebieg (gdyby jakieś auto jeździło non stop z prędkością 100 km na godzinę, to po 100 latach miałoby przebieg ponad 85 000 000 km).

Dane bardzo często są zanieczyszczone a graficzne prezentacje pozwalają nam to łatwo zauważyć.

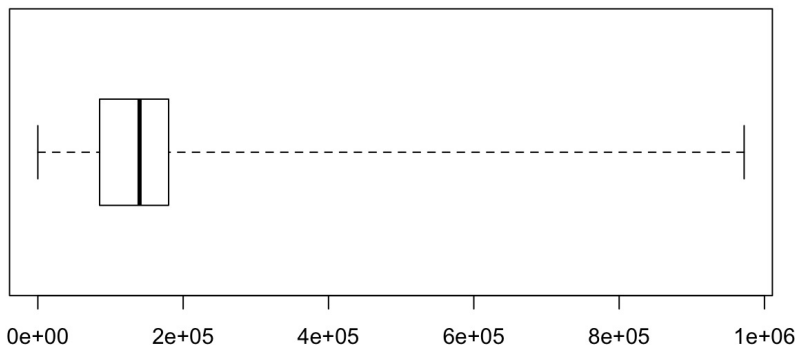
Tak jest i w tym przypadku. Oczyszczmy te dane, usuwając wszystkie wiersze, dla których cecha `Przebieg.w.km` przyjmuje wartości powyżej 1 000 000 km. Użyjemy warunku logicznego, aby indeksować tylko te wiersze o sensownym przebiegu. Więcej o tym jak działa to indeksowanie znaleźć można w odcinku 7.

```
auta2012wybrane <- auta2012[auta2012$Przebieg.w.km < 1000000,]
summary(auta2012wybrane$Przebieg.w.km)
```

##	Min.	1st Qu.	Median	Mean	3rd Qu.	I
----	------	---------	--------	------	---------	---

## 1 85000 140000 132900 180000 972

```
boxplot(auta2012wybrane$Przebieg.w.km, horizontal=TRUE)
```



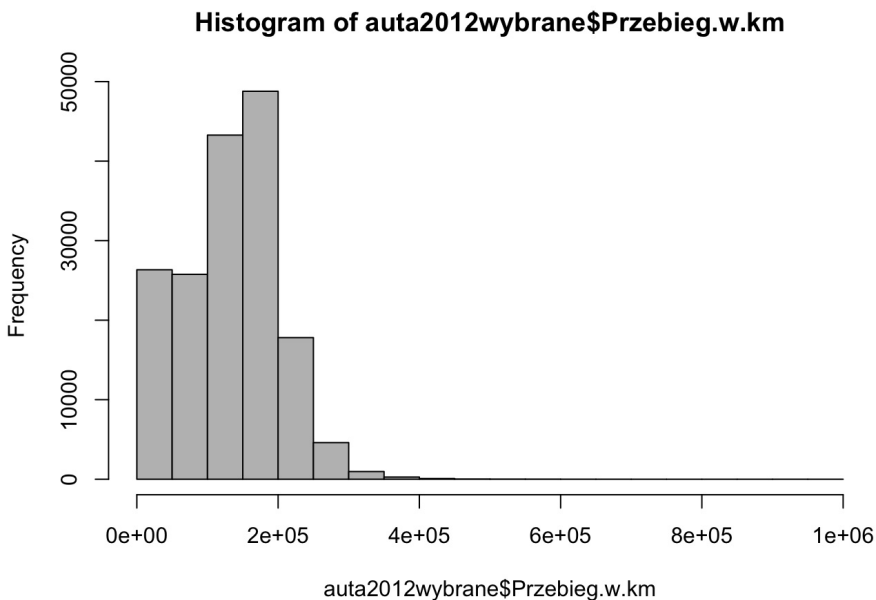
## Graficzne statystyki opisowe - histogram

Każdy z elementów tego wykresu (lewy wąs, lewa część pudełka, prawa część pudełka, prawy wąs) przedstawia 1/4 danych. Z wykresu tego możemy łatwo odczytać, że 3/4 danych dotyczy aut o przebiegu poniżej 200 tys. km. Jedynie 1/4 danych ma wyższe wartości i dla tych aut rozpiętość jest bardzo duża.

Jeżeli chcemy zaprezentować bardziej szczegółowe dane o przebiegu, możemy wykorzystać histogram. Histogram przedstawia liczbę obserwacji o określonych przedziałach wartości. Domyślnie przedziały są równo szerokie, a ich liczbę wybiera algorytm uwzględniający zmienność cechy (zazwyczaj jest to od 6 do 10 przedziałów).

Aby wykonać w programie R histogram, można wykorzystać funkcję `hist()`. Pierwszy argument określa dane, które mają być pokazane. Na poniższym przykładzie słupki zamalowano na szaro by były bardziej widoczne.

```
hist(auta2012wybrane$Przebieg.w.km, col="grey")
```



# Graficzne statystyki opisowe - histogram

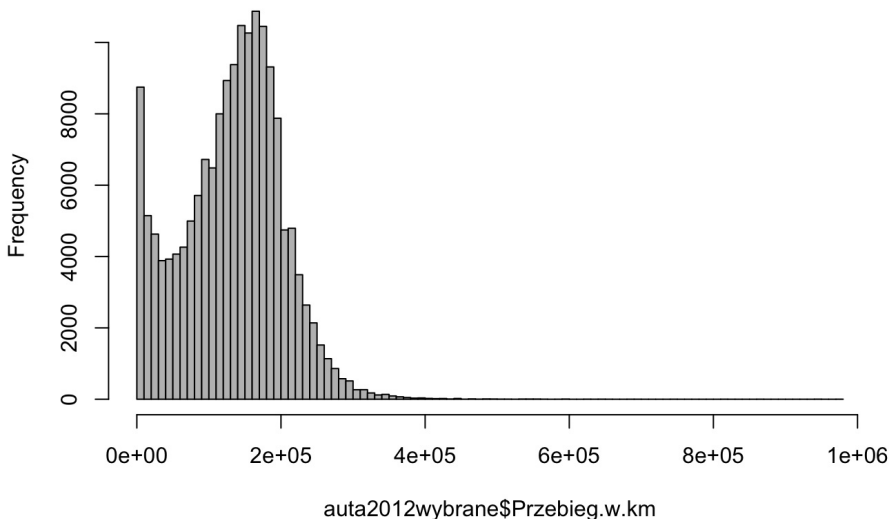
Argumentem `breaks` można określić liczbę przedziałów lub nawet granice przedziałów. Jeżeli mamy dużo obserwacji to często można więcej odczytać mając dużą liczbę przedziałów. W każdym przedziale histogram przedstawia liczbę obserwacji. Im wyższy słupek, tym więcej było aut o określonym przebiegu.

Na poniższym wykresie zauważyć można dwa „pagórki”. Jeden w pobliżu 0 oraz drugi w pobliżu 180 tys. km. Oznacza to, że w zbiorze danych jest jedna duża grupa aut o bardzo małym przebiegu, aut o przebiegu w okolicy 50 tys. km jest mniej, aut o przebiegu w okolicy 150 tys. km jest bardzo dużo, a aut o przebiegu ponad 300 tys. km jest już bardzo mało, pojedyncze sztuki.

Jak widzimy zmienną ilościową możemy opisać z różnym poziomem szczegółowości. Im większa szczegółowość tym więcej elementów na wykresie. Im więcej elementów tym więcej możemy odczytać, ale jest to też trudniejsze.

```
hist(auta2012wybrane$Przebieg.w.km, breaks = 10)
```

Histogram of auta2012wybrane\$Przebieg.w.km



---

## Graficzne statystyki opisowe - wykres punktowy

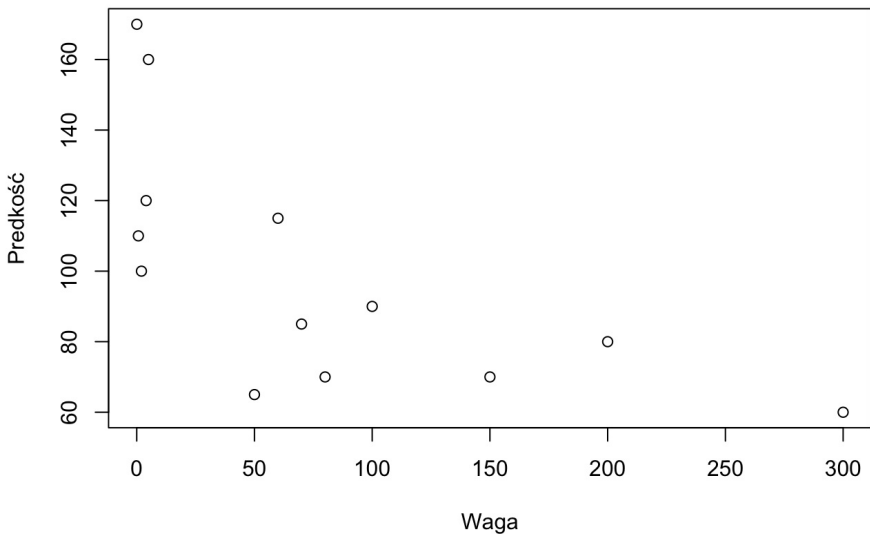
A czy można przedstawić dwie zmienne ilościowe i ich wspólną relację?

Do tego służy zazwyczaj wykres punktowy. Funkcja `plot()` jeżeli otrzyma jako pierwsze dwa argumenty cechy ilościowe to narysuje taki wykres. Dodatkowe argumenty, takie jak `xlab` i `ylab` pozwalają na określenie

nazw osi OX i OY.

Poniższy przykład pokazuje zależność pomiędzy wagą a prędkością gatunków ze zbioru `koty_ptaki`. Jak widać duża masa nie sprzyja wielkim prędkościom.

```
plot(koty_ptaki$waga, koty_ptaki$predkosc, ylab="Prędkość", xlab="Waga")
```



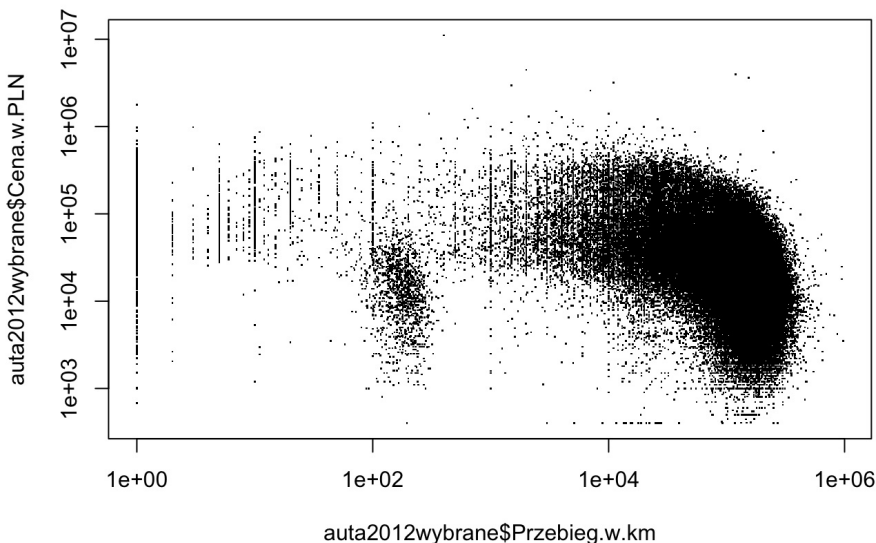
---

## Graficzne statystyki opisowe - wykres punktowy

Użyjemy funkcji `plot()` dla danych o autach, np. by pokazać zależność pomiędzy przebiegiem a ceną. Okazuje się jednak, że ponieważ ofert sprzedaży aut jest bardzo wiele, lepiej nie rysować każdej oferty dużym kołem, ale pojedynczym punktem. Można to osiągnąć dodając argument `pch="."`, którym możemy zmieniać znak za pomocą którego rysowane są na wykresie punkty odpowiadające danym. Ponieważ i przebieg i cena auta zawierają pojedyncze bardzo duże wartości zamiast pokazywać je w standardowej skali, znacznie lepiej przedstawić je w skali logarytmicznej. Można to osiągnąć dodając do wykresu argument `log="xy"`. Wartość `"xy"` oznacza, że obie osie należy zlogarytmować.

Więcej informacji o wykresach rozrzutu i innych sposobach przedstawienia cechy ilościowej przedstawionych jest w sezonie 3.

```
plot(auta2012wybrane$Przebieg.w.km, auta2012wyl
```



## Zadania:

- Przedstaw graficznie za pomocą wykresu pudełkowy oraz histogramu rozkład cechy `Cena.w.PLN`.
- Zwróć uwagę, że pojedyncze auta o bardzo wysokich cenach, powodują że wykres jest mało czytelny. Oczyszczyć te dane, pozostawiając tylko auta o cenie poniżej 100 tys. pln. Następnie przedstawić rozkład cen aut w segmencie aut do 100 tys. pln.



- Aut w jakiej cenie jest najwięcej wśród zebranych ogłoszeń?

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Cechy jakościowe

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 12*

*pogRomcy danych*

- [Co to znaczy: cecha jakościowa](#)
- [Wczytanie danych](#)
- [Cechy jakościowe](#)
- [Cechy jakościowe](#)
- [Wczytanie danych](#)
- [Statystyki opisowe](#)
- [Zadania:](#)
- [Procenty](#)
- [Procenty](#)
- [Procenty posortowane](#)
- [Procenty posortowane](#)
- [Zadania:](#)
- [Graficzne statystyki opisowe](#)
- [Graficzne statystyki opisowe](#)
- [Zadania:](#)
- [Napisy czy czynniki?](#)
- [Napisy czy czynniki?](#)
- [Napisy czy czynniki?](#)

- [Tablice częstości](#)
- [Tablice częstości](#)
- [Przekształcanie zmiennych ilościowych w jakościowe](#)
- [Tablice częstości](#)
- [Kolejność i nazwy czynników](#)
- [Kolejność i nazwy czynników](#)
- [Kolejność i nazwy czynników](#)
- [Kolejność i nazwy czynników](#)
- [Zadania:](#)

Analizując dane będziemy spotykać się z różnymi rodzajami zmiennych. Bardzo często będą to cechy jakościowe, w niektórych sytuacjach będziemy chcieli cechę ilościową przekształcić w jakościową.

W tym odcinku nauczymy się:

- jakie zmienne / cechy określa się terminem *cechy jakościowe*,
- jakie podstawowe operacje można wykonywać na cechach jakościowych,
- jak podsumowywać / opisywać cechy jakościowe.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012`, oba dostępne w pakiecie `PogromcyDanych`.

---

# Co to znaczy: cecha jakościowa

Cechy jakościowe to takie, które opisują przynależność do grup. Z tego powodu często stosuje się też nazwę *zmienna grupująca*. A ponieważ w programie R taką zmienną oznacza się terminem `factor`, to w języku polskim spotyka się kalkę *zmienna faktor*. Inną nazwą, stosowaną często przez programistów, jest *zmienna czynnikowa*.

Określając zmienną jakościową warto wskazać o jakich grupach mówimy, przykładami zmiennych jakościowych są np. wykształcenie (grupy: podstawowe, średnie, wyższe), płeć (grupy: kobieta, mężczyzna), kraj zamieszkania (grupy: Polska, ...).

Zmienna jakościowa może być opisana liczbami, jeżeli jednak te liczby nie oznaczają ilości czy wielkości to taka zmienna jest w istocie zmienną jakościową. Przykładowo pierwsze dwie cyfry kodu pocztowego mogą być zmienną jakościową, ocenę szkolną można traktować jako zmienną jakościową (można też jako ilościową zależnie od kontekstu), podobnie liczbę drzwi samochodu, jeżeli chcemy jej użyć do pogrupowania samochodów na 3 i 5 drzwiowe, potraktujemy jako cechę jakościową.

Jakkolwiek zmienne jakościowe mogą bardzo różnić się

liczbą grup oraz rodzajem grup, można na nich wykonywać kilka wspólnych operacji. Przyjrzyjmy się im.

---

## Wczytanie danych

Dane od których rozpoczniemy przykłady to `koty_ptaki` z pakietu `PogromcyDanych`. Aby te dane wczytać, wystarczy włączyć ten pakiet - instrukcja jak to zrobić znajduje się w odcinku 2.

Włączmy pakiet i użyjmy funkcji `head()` by wyświetlić pierwsze sześć wierszy. Które z tych zmiennych to zmienne jakościowe?

```
library(PogromcyDanych)
head(koty_ptaki)
```

##	gatunek	waga	dlugosc	predkosc	habitat
## Tygrys	Tygrys	300	2.5	60	Azja
## Lew	Lew	200	2.0	80	Afryka
## Jaguar	Jaguar	100	1.7	90	Ameryka
## Puma	Puma	80	1.7	70	Ameryka
## Leopard	Leopard	70	1.4	85	Azja
## Gepard	Gepard	60	1.4	115	Afryka

Każdy wiersz opisuje jeden gatunek. Zmienne jakościowe to w poniższym przypadku wszystkie te, które nie są liczbowymi, a więc *gatunek*, *habitat* i *druzyna*.

---

# Cechy jakościowe

Przjrzyjmy się dwóm z tych zmiennych - *gatunek* i *druzyna*.

```
koty_ptaki$gatunek

##      [1] "Tygrys"          "Lew"              "Jag"
##      [5] "Leopard"         "Gepard"           "Irb:
##      [9] "Strus"           "Orzel przedni"    "Sok
##     [13] "Albatros"
```

```
koty_ptaki$druzyna

##      [1] Kot   Kot   Kot   Kot   Kot   Kot   Kot   Pta
## Levels: Kot Ptak
```

Gdy te zmienne są wyświetlane, poza wartościami dodatkowo wyświetlany jest zbiór wszystkich możliwych wartości. Ten zbiór nazywa się najczęściej poziomami zmiennej jakościowej lub słownikiem zmiennej.

Dwie podstawowe operacje na zmiennych jakościowych to odczytywanie poziomów zmiennej, co można zrobić funkcją `levels()`, oraz wyznaczanie liczebności poszczególnych poziomów, co można wykonać funkcją `table()`.

To czy dana zmienna jest zmienną jakościową czy

ilościową można sprawdzić używając funkcji `class()`. Jeżeli wynikiem jest napis `factor` to mamy do czynienia ze zmienną jakościową. Jeżeli wynikiem jest `integer` lub `numeric` to mamy do czynienia ze zmienną ilościową.

```
class(koty_ptaki$druzyna)
```

```
## [1] "factor"
```

```
class(koty_ptaki$predkosc)
```

```
## [1] "integer"
```

```
class(koty_ptaki$dlugosc)
```

```
## [1] "numeric"
```

---

## Cechy jakościowe

Przykładowo dla dwóch wybranych zmiennych ze zbioru `koty_ptaki` otrzymujemy takie wyniki dla obu funkcji.

Wynikiem funkcji `levels()` jest wektor napisów - poziomów zmiennej jakościowej.

```
levels(koty_ptaki$gatunek)
```

```
## NULL
```

```
levels(koty_ptaki$druzyna)
```

```
## [1] "Kot" "Ptak"
```

Wynikiem funkcji `table()` jest wektor liczebności każdego czynnika.

```
table(koty_ptaki$gatunek)
```

```
##  
##      Albatros      Gepard      Irbis  
##           1           1  
##      Leopard      Lew      Orzel przedn  
##           1           1  
## Sokol wedrowný      Strus      Tygrys  
##           1           1
```

```
table(koty_ptaki$druzyna)
```

```
##  
##   Kot  Ptak  
##    7    6
```

Obie zmienne znacznie się różnią. Druzyna występuje tylko w dwóch wartościach a gatunek w trzynastu (każdy wiersz to inna wartość).

---

## Wczytanie danych

Zbiór danych `koty_ptaki` składa się z 13 wierszy. Można cały ten zbiór danych wyświetlić na ekranie. Nie zawsze potrzebujemy więc specjalnych statystyk opisowych by



zrozumieć co się dzieje w takich małych zbiorach danych.

Dlatego dalsze ćwiczenia ze zmiennymi jakościowymi przeprowadzimy na znaczenie większym zbiorze danych `auta2012` o ponad 200 tysiącach wierszy, który również znajduje się w pakiecie `PogromcyDanych`.

Opis tego zbioru danych znaleźć można w odcinku [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

Wczytajmy ten zbiór danych i przyjrzyjmy się dwóm pierwszym wierszom.

```
library(PogromcyDanych)
head(auta2012, 2)

## Source: local data frame [2 x 28]
##
##      Cena Waluta Cena.w.PLN Brutto.netto  KM
## 1 49900     PLN      49900      brutto 140
## 2 88000     PLN      88000      brutto 156
## Variables not shown: Liczba.drzwi (fctr), Przebieg.w.km (dbl), Rodzaj.paliwa (fctr),
##      (fctr), Kraj.aktualnej.rejestracji (fctr), Pojazd.uszkodzony (fctr), Skrzynia.biegow
##      Status.pojazdu.sprawadzonego (fctr), Wypos
##      Rodzaj.paliwa.posortowany (fctr), Kolor_na
##      Wyposazenie.dodatkowe_napis (chr), czy_me
##      (lgl), szyby (lgl), MarkaModel (chr)
```

Które z tych zmiennych to zmienne jakościowe?

Gdy w zbiorze danych jest wiele kolumn, to często wygodnie na pierwsze wiersze spojrzeć poprzez funkcję `glimpse()` która kolejne kolumny ze zbioru danych przedstawia jedna pod drugą.

```
glimpse(auta2012)

## Observations: 207602
## Variables:
##   $ Cena                                (dbl) 49900,
##   $ Waluta                              (fctr) PLN, 1
##   $ Cena.w.PLN                         (dbl) 49900,
##   $ Brutto.netto                       (fctr) brutto
##   $ KM                                  (dbl) 140, 15
##   $ kW                                  (dbl) 103, 11
##   $ Marka                              (fctr) Kia, 1
##   $ Model                             (fctr) Carens
##   $ Wersja                            (fctr) , , ,
##   $ Liczba.drzwi                      (fctr) 4/5, 4
##   $ Pojemnosc.skokowa                  (dbl) 1991, 2
##   $ Przebieg.w.km                     (dbl) 41000,
##   $ Rodzaj.paliwa                     (fctr) H, H,
##   $ Rok.produkcji                     (dbl) 2008, 2
##   $ Kolor                             (fctr) , , ,
##   $ Kraj.aktualnej.rejestracji        (fctr) Polska
##   $ Kraj.pochodzenia                  (fctr) , , ,
##   $ Pojazd.uszkodzony                 (fctr) , , ,
##   $ Skrzynia.biegow                   (fctr) manual
##   $ Status.pojazdu.sprowadzonego     (fctr) , , ,
##   $ Wyposazenie.dodatkowe             (fctr) ABS, 6
##   $ Rodzaj.paliwa.posortowany         (fctr) H, H,
##   $ Kolor_napis                       (chr) "", "",
##   $ Wyposazenie.dodatkowe_napis       (chr) "ABS, 6
##   $ czy_metallic                      (lgl) FALSE,
```

```
## $ maKlimatyzacje (lgl) TRUE, F
## $ szyby (lgl) TRUE, F
## $ MarkaModel (chr) "Kia: C
```

---

## Statystyki opisowe

Przjrzyjmy się teraz takim cechom jak `Waluta` lub `Marka`. Cechy te są opisane przez wartości z określonej listy możliwości. Dla cechy `Waluta` lista możliwości to `levels(auta2012$Waluta)=CHF, CZK, EEK, EUR, GBP, HUF, PLN, USD`.

Dla cech jakościowych możemy wykonać kilka operacji. Jedną z nich jest wyświetlenie listy możliwych wartości. Taką listę nazywa się często słownikiem (spisem możliwych wartości).

Listę możliwych wartości dla zmiennej jakościowej można wyznaczyć funkcją `levels()`.

```
levels(auta2012$Waluta)

## [1] "CHF" "CZK" "EEK" "EUR" "GBP" "HUF" "PLN"
```

Zmienne jakościowe często opisuje się tablicą liczebności, a więc informacją ile razy wystąpiła każda z wartości ze słownika. Tablicę liczebności można wyznaczyć funkcją `table()` lub `summary()`. Jeżeli w

zmiennej jakościowej występują wartości brakujące, to funkcja `summary()` też napisze ile ich jest (funkcja `table()` domyślnie tego nie robi, można takie zachowanie wymusić dodając argument `use.NA = "always"`).

```
table(auta2012$Waluta)
```

```
##
```

##	CHF	CZK	EEK	EUR	GBP	HUF
##	5	19070	2	5407	5	611 18

```
summary(auta2012$Waluta)
```

##	CHF	CZK	EEK	EUR	GBP	HUF
##	5	19070	2	5407	5	611 18

## Zadania:

- Ile zmiennych w zbiorze danych `auta2012` to zmienne jakościowe?
- Zmienna `Liczba.drzwi` przyjmuje wartości `2/3` i `4/5`. Zachowuje się ona jako zmienna jakościowa. Jednak można uznać, że liczba drzwi to cecha ilościowa, ponieważ opisuje *ilość* drzwi w samochodzie. Jak wyjaśnić tę dualność?

Przykładowe odpowiedzi znajdują się na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

---

# Procenty

W pewnych sytuacjach, zamiast posługiwać się tablicą liczebności, wygodniej jest posługiwać się tabelą z procentami.

Ale jak te procenty policzyć?

Można to zrobić na kilka sposobów. Funkcja `prop.table()` dzieli wartości tabeli liczebności przez sumę liczebności, a więc zamienia liczebności na frakcje (frakcje czyli procenty, części całości). Przypiszmy wynik funkcji `table()` do jakiejś zmiennej, np. o nazwie `waluty` (nazwa może być dowolna, musimy jej jednak konsekwentnie używać w dalszych przykładach). Następnie tak otrzymaną tabelę liczebności zamieniamy na frakcje.

W poniższym przykładzie, instrukcję przypisania wyniku obejmujemy nawiasami, po to by jej wynik był wyświetlony. Bez nawiasów wynik funkcji `table()` nie zostałby wyświetlony, a jedynie przypisany do zmiennej `waluty`.

```
(waluty <- table(auta2012$Waluta))
```

```
##
```

##	CHF	CZK	EEK	EUR	GBP	HUF
##	5	19070	2	5407	5	611 18

```
prop.table(waluty)
```

```
##
##              CHF              CZK              EEK
## 2.408455e-05 9.185846e-02 9.633819e-06 2.604
##              HUF              PLN              USD
## 2.943132e-03 8.759261e-01 3.169526e-03
```

## Procenty

Funkcja `prop.table()` wyznacza frakcje, ale nie wyglądają one zbyt czytelnie dla człowieka. Wygodniej by było wypisać procenty. I to nie byłyby jakie procenty, ale procenty które są ładnie sformatowane, np. wyświetlone z dokładnością do jednego miejsca dziesiętnego po przecinku.

Aby zamienić frakcje na procenty, wystarczy je przemnożyć przez 100, a aby wynik przedstawić z dokładnością do jednego miejsca po przecinku wykorzystamy funkcję `round()`, która zaokrągla liczby do wskazanej liczby miejsc po kropce dziesiętnej.

```
waluty <- table(auta2012$Waluta)
(fracje <- prop.table(waluty))
```

```
##
```

```
##          CHF          CZK          EEK
## 2.408455e-05 9.185846e-02 9.633819e-06 2.604
##          HUF          PLN          USD
## 2.943132e-03 8.759261e-01 3.169526e-03
```

```
(procenty <- 100*frakcje)
```

```
##
##          CHF          CZK          EEK
## 2.408455e-03 9.185846e+00 9.633819e-04 2.604
##          HUF          PLN          USD
## 2.943132e-01 8.759261e+01 3.169526e-01
```

```
round(procenty, digits = 1)
```

```
##
##  CHF  CZK  EEK  EUR  GBP  HUF  PLN  USD
##  0.0   9.2  0.0  2.6  0.0  0.3 87.6  0.3
```

Wszystkie przedstawione powyżej operacje działają i na pojedynczych wartościach jak i na wektorach.

Przykładowo operacja `100*frakcje` wymnaża liczbę 100 z każdym elementem wektora `frakcje` i wynikiem tej operacji jest wektor. Nie w każdym języku programowania można mieć takie mieszane operacje mnożenia wektorów i pojedynczych liczb. W programie R jest to możliwe ponieważ od podstaw wszystkie operacje były planowane jako operacje wektorowe i nawet pojedyncze wartości są w rzeczywistości wektorami o długości 1. Wykonując operacje na dwóch wektorach krótszy jest powtarzany tak długo aż długością zrówna się z dłuższym.

---

# Procenty posortowane

Mamy już procenty. Aby dodatkowo zwiększyć czytelność tej formy prezentacji warto te procenty posortować rosnąco lub malejąco.

Nawet dla ośmiu liczb, jeżeli każda z nich jest opisana przez wiele cyfr, czasem w gąszczu cyfr trudno nawet dostrzec, która jest największa.

Do sortowania można wykorzystać funkcję `sort()`. Dodając argument `decreasing = TRUE` powodujemy, że wartości będą uporządkowane malejąco.

```
zaokragloneProcenty <- round(procenty,1)
sort(zaokragloneProcenty)
```

##								
##	CHF	EEK	GBP	HUF	USD	EUR	CZK	PLN
##	0.0	0.0	0.0	0.3	0.3	2.6	9.2	87.6

```
sort(zaokragloneProcenty, decreasing = TRUE)
```

##								
##	PLN	CZK	EUR	HUF	USD	CHF	EEK	GBP
##	87.6	9.2	2.6	0.3	0.3	0.0	0.0	0.0

---

# Procenty posortowane



Wartości w wektorze można posortować też na kilka innych sposobów. Bardziej skomplikowanym, ale pozwalającym na omówienie bardziej zaawansowanych przykładów indeksowania jest funkcja `order()`.

Wynikiem funkcji `order()` są indeksy elementów w wektorze, które tworzą ciąg rosnący. Więcej informacji o tej funkcji znaleźć można na śladzie *Sortowanie przez indeksowanie* w odcinku 7.

```
order(zaokragloneProcenty)
```

```
## [1] 1 3 5 6 8 4 2 7
```

Jeżeli wektor `zaokragloneProcenty` będziemy czytać w kolejności, pierwszy element, trzeci, piąty, szósty, ósmy, czwarty, drugi i siódmy, to otrzymamy posortowany ciąg.

Możemy więc wynik tej funkcji wykorzystać do indeksowania oryginalnego zbioru danych, i w ten sposób posortować cały wektor.

```
kolejnoscPosortowanych <- order(zaokragloneProcenty)
zaokragloneProcenty[kolejnoscPosortowanych]
```

```
##
##   CHF   EEK   GBP   HUF   USD   EUR   CZK   PLN
##   0.0   0.0   0.0   0.3   0.3   2.6   9.2  87.6
```

Funkcja `rev()` odwraca kolejność elementów w

wektorze, możemy jej użyć by posortować wartości malejąco.

```
zaokrągloneProcenty[rev(kolejnoscPosortowanych)]
```

```
##
```

##	PLN	CZK	EUR	USD	HUF	GBP	EEK	CHF
##	87.6	9.2	2.6	0.3	0.3	0.0	0.0	0.0

A jak sprawdzić, na których pozycjach występują trzy najmniejsze wartości?

```
kolejnoscPosortowanych[1:3]
```

```
## [1] 1 3 5
```

---

## Zadania:

- Cecha `Marka` opisuje markę samochodu. Sprawdź, która marka jest najpopularniejsza.
- Cecha `Rodzaj.paliwa` opisuje rodzaj paliwa wykorzystywanego przez auto. Czy jest to benzyna, olej, gaz? Sprawdź jaki procent samochodów jest napędzanych na benzynę.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

---

# Graficzne statystyki opisowe

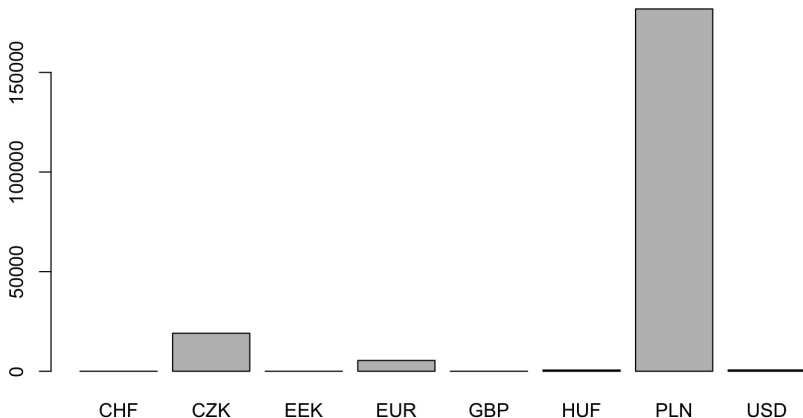
Tabele liczebności są proste w interpretacji. Podobnie jednak jak w innych przypadkach, graficzne przedstawienie liczb pozwala na łatwiejsze dostrzeżenie co się dzieje w danych. Koniec końców, w tabeli liczb łatwo pomylić się nawet gdy chodzi o liczbę cyfr w liczbie. Na wykresie takie wartości natychmiast rzucają się w oczy.

Zazwyczaj tabele liczebności przedstawia się za pomocą wykresów słupkowych. Wykresy słupkowe można wykonać np. funkcją `barplot()`.

```
(waluty <- table(auta2012$Waluta))
```

##							
##	CHF	CZK	EEK	EUR	GBP	HUF	
##	5	19070	2	5407	5	611	18

```
barplot(waluty)
```



---

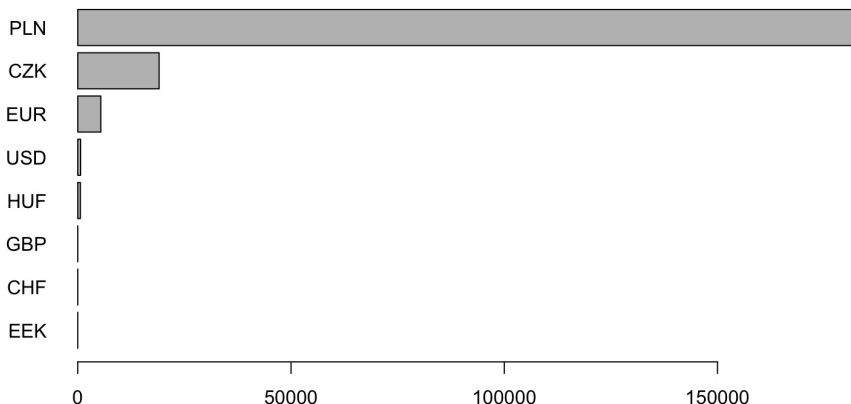
## Graficzne statystyki opisowe

Jeżeli chcemy, by słupki były przedstawiane poziomo, do funkcji `barplot()` można dodać argument `horiz = TRUE`.

Domyślnie wartości są uporządkowane alfabetycznie. Nie zawsze jednak taka kolejność ma sens. W tym przypadku rozsądniej jest posortować dane malejąco, co można wykonać z użyciem funkcji `sort()`.

Argument `las=1` powoduje, że oś OY ma etykiety ustawione poziomo.

```
posortowaneWaluty <- sort(waluty)
barplot(posortowaneWaluty, horiz = TRUE, las=1)
```



## Zadania:

- Przedstaw graficznie tabelę liczebności dla zmiennej `Kraj.pochodzenia`. Wypróbuj wykres pionowy i poziomy. Aby obrócić kierunek etykiet na osiach dodaj do funkcji `barplot()` argument `las=1`.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

# Napisy czy czynniki?

Z wyglądu, cechy jakościowe (czynniki) przypominają cechy napisowe. Jednak pod spodem, ich reprezentacja jest zupełnie inna. Czynniki nie są pamiętane jako napisy, ale jako para - wektor liczb i słownik, określający, który napis odpowiada, której liczbie.

Na napisach zazwyczaj wykonuje się inne operacje niż na zmiennych jakościowych, może się więc tak zdarzyć, że celowo chcemy jakieś dane odczytać jako napisy. Ale gdy dane są wczytywane z pliku tekstowego w którym są napisy, domyślnie traktowane są jako czynniki.

W poniższym przykładzie takie zmienne jak `gatunek` i `habitat` są wczytywane jako zmienna klasy `factor`. Efektem ubocznym jest sposób wyświetlania wyników np. przez zmienną `cat()`. Dla zmiennej czynnikowej, ta funkcja na ekranie wypisze liczby a nie wartości napisowe.

```
koty_ptaki <- read.table("http://biecek.pl/MOOC
                        sep=";", dec=".", header=TRUE)
str(koty_ptaki)
```

```
## 'data.frame':    13 obs. of  7 variables:
##  $ gatunek      : Factor w/ 13 levels "Albatros"
##  $ waga         : num  300 200 100 80 70 60 50
##  $ dlugosc      : num  2.5 2 1.7 1.7 1.4 1.4 1.1
```

```
## $ predkosc : int 60 80 90 70 85 115 65 170
## $ habitat : Factor w/ 6 levels "Afryka", "Azja", "Ameryka", "Australia", "Antarktyda", "Północna Ameryka"
## $ zywnosc : int 25 29 15 13 21 12 18 20
## $ druzyna : Factor w/ 2 levels "Kot", "Ptak"
```

```
class(koty_ptaki$gatunek)
```

```
## [1] "factor"
```

```
cat(koty_ptaki$gatunek)
```

```
## 13 7 4 9 6 2 3 5 12 8 11 10 1
```

---

## Napisy czy czynniki?

Funkcja `read.table()` posiada argument `stringsAsFactors` określający w jaki sposób wczytywać kolumny z napisami. Ustawivszy argument `stringsAsFactors=FALSE` wymuszamy wczytywanie danych jako zmienne napisowe.

Zauważmy jak teraz wygląda wynik funkcji `str()` i `cat()`.

```
koty_ptaki <- read.table("http://biecek.pl/MOODS/koty_ptaki.csv",
  sep=";", dec=",", header=TRUE, stringsAsFactors=FALSE)
str(koty_ptaki)
```

```
## 'data.frame': 13 obs. of 7 variables:
## $ gatunek : chr "Tygrys" "Lew" "Jaguar" "Pantera" "Koczkodziej" "Koczkodziej" "Koczkodziej"
## $ waga : num 300 200 100 80 70 60 50 40 30 20 10 5 2
```

```
## $ dlugosc : num 2.5 2 1.7 1.7 1.4 1.4 1.4
## $ predkosc : int 60 80 90 70 85 115 65 170
## $ habitat : chr "Azja" "Afryka" "Ameryka"
## $ zywnosc : int 25 29 15 13 21 12 18 20
## $ druzyna : chr "Kot" "Kot" "Kot" "Kot"
```

```
class(koty_ptaki$gatunek)
```

```
## [1] "character"
```

```
cat(koty_ptaki$gatunek)
```

```
## Tygrys Lew Jaguar Puma Leopard Gepard Irbis
```

Tym razem kolumna `gatunek` została wczytana jako wektor napisów.

## Napisy czy czynniki?

Zmienne czynnikowe można konwertować na napisy funkcją `as.character()`.

Napisy można konwertować na zmienne czynnikowe funkcją `as.factor()`.

Przedstawmy działanie obu funkcji na przykładzie.

```
czynniki <- as.factor(koty_ptaki$gatunek)
str(czynniki)
```

```
## Factor w/ 13 levels "Albatros","Gepard",...
```



```
napisy <- as.character(czynniki)
str(napisy)
```

```
## chr [1:13] "Tygrys" "Lew" "Jaguar" "Puma" "
```

---

## Tablice częstości

A co gdybyśmy chcieli przedstawić zależność pomiędzy dwoma zmiennymi jakościowymi? Można to zrobić używając funkcji `table()` (należy kolejne zmienne podawać jako kolejne argumenty). Przykładowe wywołanie dla dwóch zmiennych: Waluty i Kraju pochodzenia.

```
table(auta2012$Kraj.pochodzenia, auta2012$Waluta)
```

Jednak wygodniej jest, takie tabele tworzyć używając funkcji `xtabs()`, która jest trochę wygodniejsza w pracy. Pierwszym argumentem tej funkcji jest formuła o składni `~ zmienna_1 + .. + zmienna_n`, t.j. rozpoczynająca się tyldą a następnie ze zmiennymi porodzielanymi znakiem plusa. Drugim argumentem jest ramka danych, która powinna zawierać zmienne wymienione w formule.

Wynikiem jest tablica kontyngencji o tylu wymiarach, ile zmiennych występuje w formule.

Przykładowo, zestawiając ze sobą kraj pochodzenia oraz

walutę w ogłoszeniu, otrzymujemy tablicę częstości o wymiarach 32 wiersze i 8 kolumn. Pierwszy wiersz tej tabeli odpowiada pustej wartości kraju pochodzenia, stąd brak nazwy w pierwszym wierszu.

```
krajWaluta <- xtabs( ~ Kraj.pochodzenia + Waluta)
krajWaluta
```

##		Waluta			
##	Kraj.pochodzenia	CHF	CZK	EEK	EUR
##		0	5096	2	3619
##	Austria	0	522	0	26
##	Belgia	0	373	0	356
##	Bulgaria	0	0	0	0
##	Chorwacja	0	0	0	0
##	Czechy	0	6099	0	0
##	Dania	0	30	0	177
##	Estonia	0	1	0	0
##	Francja	0	987	0	37
##	Grecja	0	0	0	0
##	Hiszpania	0	23	0	35
##	Holandia	0	81	0	145
##	Irlandia	0	0	0	1
##	Islandia	0	0	0	0
##	Kanada	0	6	0	1
##	Litwa	0	0	0	0
##	lotwa	0	0	0	0
##	Luksemburg	0	13	0	0
##	Monako	0	0	0	0
##	Niemcy	1	4194	0	977
##	Norwegia	0	0	0	0
##	Polska	1	1	0	12
##	Rosja	0	1	0	0
##	Rumunia	0	1	0	0

##	Slowacja	0	36	0	4
##	Stany Zjednoczone	0	150	0	4
##	Szwajcaria	3	159	0	2
##	Szwecja	0	7	0	0
##	Ukraina	0	1	0	0
##	Węgry	0	0	0	0
##	Wielka Brytania	0	14	0	0
##	Włochy	0	1275	0	11

Funkcja `prop.table()` liczy procenty (z drugim argumentem `=1` liczy procenty w wierszach) a funkcja `round()` zaokrągla liczby (z dokładnością do tylu cyfr po kropce ile wskazano w drugim argumencie).

```
round(prop.table(krajWaluta, 1), 3)
```

##		Waluta			
##	Kraj.pochodzenia	CHF	CZK	EEK	EUR
##		0.000	0.057	0.000	0.041
##	Austria	0.000	0.263	0.000	0.013
##	Belgia	0.000	0.065	0.000	0.062
##	Bulgaria	0.000	0.000	0.000	0.000
##	Chorwacja	0.000	0.000	0.000	0.000
##	Czechy	0.000	0.990	0.000	0.000
##	Dania	0.000	0.051	0.000	0.302
##	Estonia	0.000	0.143	0.000	0.000
##	Francja	0.000	0.111	0.000	0.004
##	Grecja	0.000	0.000	0.000	0.000
##	Hiszpania	0.000	0.072	0.000	0.110
##	Holandia	0.000	0.022	0.000	0.039
##	Irlandia	0.000	0.000	0.000	0.037
##	Islandia	0.000	0.000	0.000	0.000
##	Kanada	0.000	0.059	0.000	0.010
##	Litwa	0.000	0.000	0.000	0.000

##	lotwa	0.000	0.000	0.000	0.000
##	Luksemburg	0.000	0.040	0.000	0.000
##	Monako	0.000	0.000	0.000	0.000
##	Niemcy	0.000	0.084	0.000	0.020
##	Norwegia	0.000	0.000	0.000	0.000
##	Polska	0.000	0.000	0.000	0.000
##	Rosja	0.000	0.077	0.000	0.000
##	Rumunia	0.000	0.250	0.000	0.000
##	Slowacja	0.000	0.655	0.000	0.073
##	Stany Zjednoczone	0.000	0.047	0.000	0.001
##	Szwajcaria	0.002	0.105	0.000	0.001
##	Szwecja	0.000	0.045	0.000	0.000
##	Ukraina	0.000	0.167	0.000	0.000
##	Węgry	0.000	0.000	0.000	0.000
##	Wielka Brytania	0.000	0.011	0.000	0.000
##	Włochy	0.000	0.401	0.000	0.003

## Tablice częstości

Tablice częstości nie są ramkami danych.

Gdy w grę wchodzi dwie zmienne, to rzeczywiście wyglądają one jak ramki danych - mają wiersze i kolumny. Muszą być jednak czymś innym, ponieważ w sytuacji, gdy tablica kontyngencji dotyczy trzech lub większej liczby zmiennych to wynikowa macierz kontyngencji ma więcej wymiarów niż dwa.

Funkcją `as.data.frame` można przekształcić wielowymiarowe tabele do ramki danych. Otrzyma się w

tym przypadku tak zwaną reprezentację rzadką macierzy kontyngencji, w której pierwsze kolumny opisują wszystkie kombinacje czynników zmiennych a ostatnia kolumna opisuje liczebność występowania poszczególnej kombinacji czynników.

```
xt <- xtabs(~Kraj.pochodzenia+Waluta, auta2012)
xt <- as.data.frame(xt)
## Wyświetlmy tylko te kombinacje, które rzeczywiście wystąpiły
xt[xt$Freq > 0, ]
```

##	Kraj.pochodzenia	Waluta	Freq
## 20	Niemcy	CHF	1
## 22	Polska	CHF	1
## 27	Szwajcaria	CHF	3
## 33		CZK	5096
## 34	Austria	CZK	522
## 35	Belgia	CZK	373
## 38	Czechy	CZK	6099
## 39	Dania	CZK	30
## 40	Estonia	CZK	1
## 41	Francja	CZK	987
## 43	Hiszpania	CZK	23
## 44	Holandia	CZK	81
## 47	Kanada	CZK	6
## 50	Luksemburg	CZK	13
## 52	Niemcy	CZK	4194
## 54	Polska	CZK	1
## 55	Rosja	CZK	1
## 56	Rumunia	CZK	1
## 57	Słowacja	CZK	36
## 58	Stany Zjednoczone	CZK	150
## 59	Szwajcaria	CZK	159
## 60	Szwecja	CZK	7

##	61	Ukraina	CZK	1
##	63	Wielka Brytania	CZK	14
##	64	Wlochy	CZK	1275
##	65		EEK	2
##	97		EUR	3619
##	98	Austria	EUR	26
##	99	Belgia	EUR	356
##	103	Dania	EUR	177
##	105	Francja	EUR	37
##	107	Hiszpania	EUR	35
##	108	Holandia	EUR	145
##	109	Irlandia	EUR	1
##	111	Kanada	EUR	1
##	116	Niemcy	EUR	977
##	118	Polska	EUR	12
##	121	Slowacja	EUR	4
##	122	Stany Zjednoczone	EUR	4
##	123	Szwajcaria	EUR	2
##	128	Wlochy	EUR	11
##	129		GBP	4
##	159	Wielka Brytania	GBP	1
##	161		HUF	543
##	173	Irlandia	HUF	1
##	180	Niemcy	HUF	3
##	186	Stany Zjednoczone	HUF	1
##	190	Węgry	HUF	63
##	193		PLN	79437
##	194	Austria	PLN	1439
##	195	Belgia	PLN	4984
##	196	Bulgaria	PLN	3
##	197	Chorwacja	PLN	1
##	198	Czechy	PLN	62
##	199	Dania	PLN	379
##	200	Estonia	PLN	6
##	201	Francja	PLN	7885

##	202	Grecja	PLN	4
##	203	Hiszpania	PLN	260
##	204	Holandia	PLN	3512
##	205	Irlandia	PLN	25
##	206	Islandia	PLN	3
##	207	Kanada	PLN	94
##	208	Litwa	PLN	10
##	209	lotwa	PLN	3
##	210	Luksemburg	PLN	308
##	211	Monako	PLN	2
##	212	Niemcy	PLN	44701
##	213	Norwegia	PLN	12
##	214	Polska	PLN	31309
##	215	Rosja	PLN	12
##	216	Rumunia	PLN	3
##	217	Slowacja	PLN	15
##	218	Stany Zjednoczone	PLN	2771
##	219	Szwajcaria	PLN	1348
##	220	Szwecja	PLN	148
##	221	Ukraina	PLN	5
##	222	Węgry	PLN	3
##	223	Wielka Brytania	PLN	1208
##	224	Włochy	PLN	1892
##	225		USD	363
##	239	Kanada	USD	1
##	244	Niemcy	USD	1
##	246	Polska	USD	1
##	250	Stany Zjednoczone	USD	291
##	251	Szwajcaria	USD	1

---

# Przekształcanie zmiennych ilościowych w jakościowe

Opisując dane ilościowe, często można ułatwić ich zrozumienie, jeżeli zmienne ilościowe podzieli się na przedziały. Takie przedziały są już zmienną jakościową.

Zmienną jakościową z ilościowej można stworzyć używając funkcji `cut()`, która za pierwszy argument przyjmuje zmienną ilościową, a jako drugi liczbę przedziałów do zbudowania lub punkty odcięcia dla tych przedziałów.

W przykładzie poniżej przekształcimy zmienną `waga` ze zbioru `koty_ptaki` na cztery przedziały: 0-1, 1-10, 10-100 i 100-1000. Tak stworzoną zmienną dodamy do zbioru `koty_ptaki` pod nazwą `wagaKategoria`.

```
koty_ptaki$waga
```

```
##      [1] 300.00 200.00 100.00  80.00  70.00  60.00
##     [11]  0.70   2.00   4.00
```

```
koty_ptaki$wagaKategoria <- cut(koty_ptaki$waga,
table(koty_ptaki$wagaKategoria))
```

```
##
##      (0,1]      (1,10]      (10,100]      (100,1e+03]
##           2           3           5
```

---

## Tablice częstości



Wykorzystajmy tę nowo utworzoną zmienną do przedstawienia graficznych statystyk dla pary zmiennych jakościowych. Zestawimy wagę ze zmienną drużyna.

Następnie dwuwymiarową tabelę prześlemy do funkcji `mosaicplot()`.

Rysuje ona tak zwany wykres mozaikowy przedstawiający jednocześnie wiele zależności. W pierwszym kroku szerokość kolumn odpowiada proporcjom wartości Kot i Ptak w zbiorze danych (jest ich prawie po równo, dlatego obie kolumny mają dosyć podobną szerokość). Następnie wysokość wierszy odpowiada względnemu udziałowi poszczególnych wartości w określonej kolumnie. W kolumnie Kot dominuje przedział dla wagi 10-100, a w kolumnie Ptak dominuje przedział 1-10 oznaczony kolorem niebieskim.

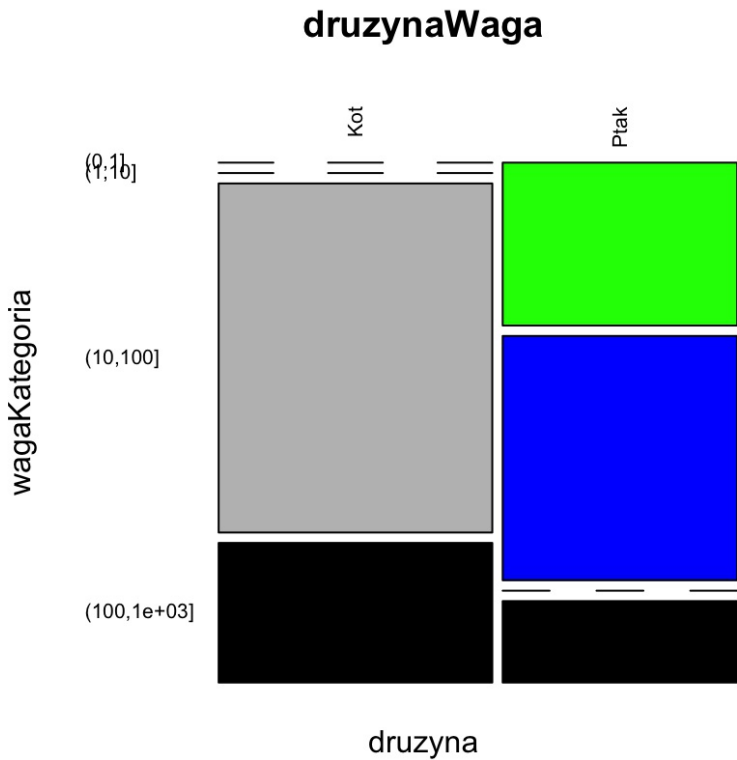
Jest to sposób na przedstawienie rozkładów warunkowych. Bardzo użyteczne, jednak wymagające pewnej wprawy w czytaniu. Więcej o wykresach mozaikowych i innych metodach przedstawiania dwóch zmiennych jakościowych znaleźć można w sezonie 3.

```
(druzynaWaga <- xtabs(~druzyna + wagaKategoria,
```

```
##           wagaKategoria
## drużyna  (0,1] (1,10] (10,100] (100,1e+03]
##      Kot      0      0          5          2
```

##	Ptak	2	3	0	1
----	------	---	---	---	---

```
mosaicplot(druzynaWaga, las=2, col=c("green", "blue", "black"))
```



---

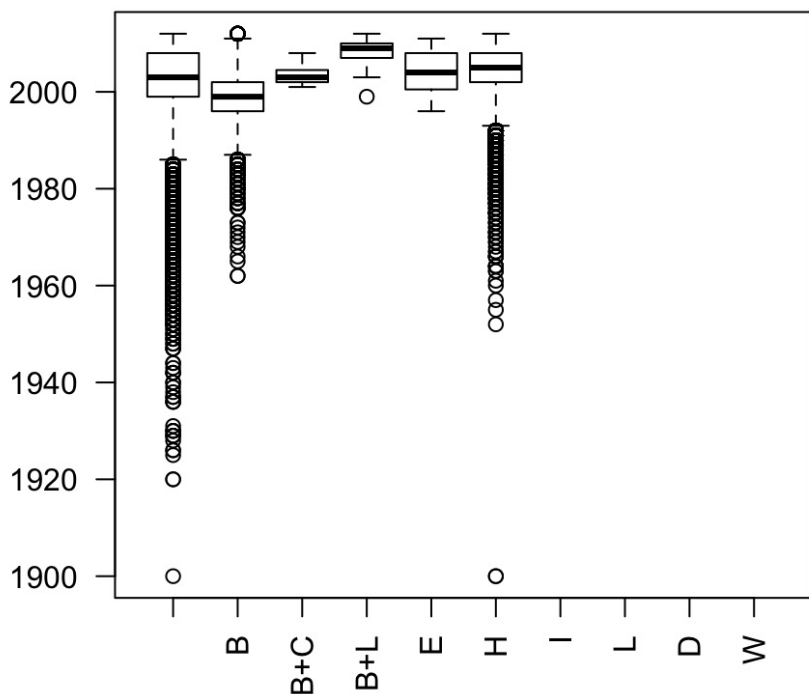
**Kolejność i nazwy czynników**

Domyślna kolejność czynników, to kolejność leksykograficzna. Tworząc wykresy, zazwyczaj wartości prezentowane są zgodnie z kolejnością opisaną przez atrybut `levels`.

```
levels(auta2012$Rodzaj.paliwa)
```

```
##      [1] ""      "B"      "B+C"    "B+L"    "E"      "H"      "I"
```

```
boxplot(Rok.produkcji ~ Rodzaj.paliwa, auta2012)
```



## Kolejność i nazwy czynników

Domyślną kolejność można zmienić. Jednym ze sposobów określania kolejności czynników jest argument `levels` w funkcji `factor()`. Czynniki będą mieć kolejność zgodną z

tam wskazaną.

```
levels(auta2012$Skrzynia.biegow)
```

```
## [1] "manualna"      ""                "automatyc
```

```
auta2012$Skrzynia.biegow <- factor(auta2012$Sk  
                                levels=c("manua  
levels(auta2012$Skrzynia.biegow)
```

```
## [1] "manualna"      ""                "automatyc
```

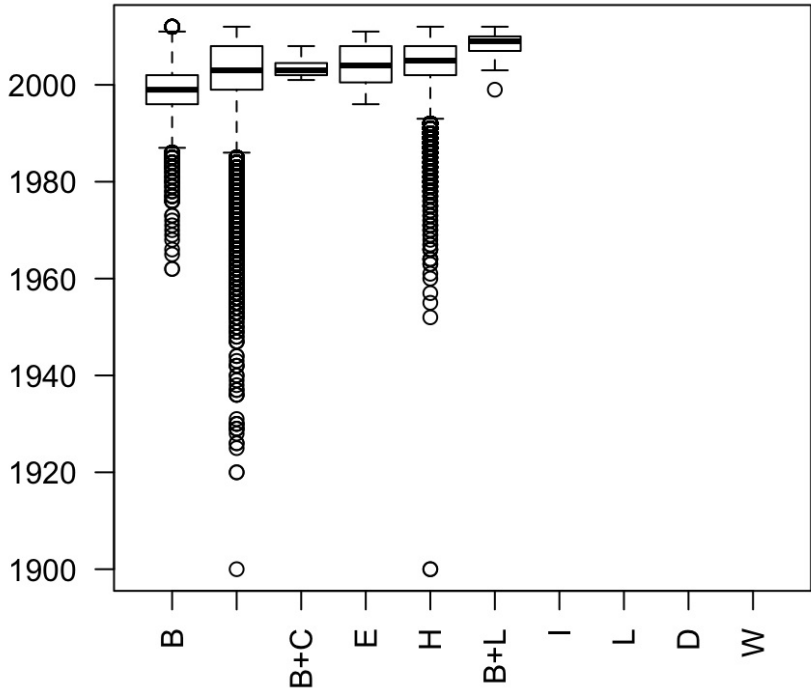
---

## Kolejność i nazwy czynników

Jeżeli chcemy „posortować” poziomy zgodnie z pewną cechą ilościową, to wygodne będzie użycie funkcji `reorder()`. Jako argumenty pobiera ona wektor ze zmienną czynnikową, wektor z dowolną zmienną i funkcję. Dla każdej grupy określonej przez zmienną czynnikową, dla wartości wskazanych przez drugi argument wyznaczana jest wartość funkcji - trzeciego argumentu. Następnie czynniki są porządkowane zgodnie z kolejnością wyników trzeciej funkcji.

W przykładzie poniżej, rodzaje paliwa są porządkowane zgodnie ze średnim rokiem produkcji aut o określonym rodzaju paliwa. Tak jak się można spodziewać, bardziej „nowoczesne” paliwa są w autach młodszych.

```
auta2012$Rodzaj.paliwa.posortowany <- reorder(
boxplot(Rok.produkcji ~ Rodzaj.paliwa.posortowa
```



# Kolejność i nazwy czynników

Nazwy czynników można też dowolnie zmieniać.

Najprościej można to zrobić używając funkcji `levels()` tak jak na poniższym przykładzie.

```
levels(auta2012$Rodzaj.paliwa)
```

```
##      [1] ""      "B"      "B+C"    "B+L"    "E"      "H"      "I"
```

```
levels(auta2012$Rodzaj.paliwa) <- c("", "B", "I")
levels(auta2012$Rodzaj.paliwa)
```

```
##      [1] ""      "B"      "B+C"    "B+L"    "E"      "H"      "I"
```

---

## Zadania:

- W zbiorze danych `auta2012` podziel zmienne *Rok.produkcji* na przedziały 1900-1990, 1991-1995, 1996-2000, 2001-2005, 2006-2010, 2011-2012, a zmienną *Przebieg.w.km* na przedziały 0-1000, 1001-10 000, 10 001-100 000, 100 001 - 1 000 000, 1 000 000 - 10 000 000.
- Wyznacz tabelę liczebności dla tych dwóch nowych zmiennych.
- Przedstaw tę tabelę graficznie.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)



# Cechy logiczne

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 13*

*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Co to znaczy: cecha logiczna](#)
- [Logika trójwartościowa](#)
- [Tabliczka logicznego dodawania i logicznego mnożenia](#)
- [Wczytanie danych](#)
- [Statystyki opisowe](#)
- [Operator negacji](#)
- [Wczytanie danych](#)
- [Statystyki opisowe](#)
- [Testy dla typów i wartości](#)
- [Logiczne 'i' oraz logiczne 'lub'](#)
- [Zadania:](#)

## O czym jest ten odcinek

Analizując dane spotkamy się z różnymi rodzajami

zmiennych. Omówiliśmy ilościowe i jakościowe. Specyficzną odmianą cech jakościowych są cechy logiczne, przyjmujące wartości logiczne prawda / fałsz.

W tym odcinku nauczymy się:

- jakie zmienne / cechy określa się terminem *cechy logiczne*,
- jakie podstawowe operacje można wykonywać na cechach logicznych,
- jak podsumowywać / opisywać cechy logiczne.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012`, oba dostępne w pakiecie `PogromcyDanych`.

---

## Co to znaczy: cecha logiczna

Cechy logiczne to takie, które opisują jedną z dwóch wartości logicznych prawda/fałsz. Przykładowo: czy wzrost jest większy niż 150 cm, czy pada deszcz, czy pacjent przeżył 5 lat po operacji itp.

Zmienne logiczne często powstają w wyniku przetwarzania zmiennych ilościowych lub jakościowych.

Mając wykształcenie opisane przez trzy poziomy, można skonstruować sztuczną cechę: czy ma wykształcenie wyższe. A mając wzrost opisany jako cecha ilościowa można skonstruować cechę logiczną, czy jest wyższy niż 150 cm.

Zmienne logiczne można traktować jak zmienne jakościowe, przyjmują bowiem wartości z dwuelementowego słownika. Podobnie jak dla wartości jakościowych, tak i dla wartości logicznych można wyznaczać tablicę liczebności.

Dla zmiennych logicznych dostępnych jest kilka dodatkowych operacji logicznych, których nie można wykonać na typowych zmiennych jakościowych.

---

## Logika trójwartościowa

Napisaaliśmy wcześniej, że wartości logiczne przyjmują wartości `TRUE` (logiczna prawda) lub `FALSE` (logiczny fałsz). Jednak równocześnie każdy rodzaj zmiennych w programie R może również przyjmować wartość nieustaloną `NA`. W gruncie rzeczy wartości logiczne operują zatem w logice trójwartościowej.

Tworząc wartości logiczne często wykorzystuje się

operatory logiczne > (większy), >= (większy równy), == (równy), <= (mniejszy równy), < (mniejszy). Gdy porównuje się takim operatorem wartość nieustaloną NA to wynik również jest wartością nieustaloną.

```
c(1, 5, 7, 3, 8, NA) <= 4
```

```
## [1] TRUE FALSE FALSE TRUE FALSE NA
```

```
c("Ala", "Ola", "Ula", NA, "Ela", "Ola") == "Ola"
```

```
## [1] FALSE TRUE FALSE NA FALSE TRUE
```

Tworząc wektory wartości logicznych zamiast pełnych nazw TRUE i FALSE można również korzystać ze skrótów T i F.

```
c(T, F, NA, TRUE, FALSE)
```

```
## [1] TRUE FALSE NA TRUE FALSE
```

Specjalnymi operatorami dla wartości logicznych są | (logiczne lub) oraz & (logiczne i). Pierwszy z nich zwraca jako wynik wartość TRUE jeżeli którykolwiek argument jest prawdziwy. Drugi zwraca jako wynik wartość TRUE jeżeli oba argumenty są prawdziwe.

```
TRUE | FALSE
```

```
## [1] TRUE
```

```
(4 > 2) | ("A" == "a")
```

```
## [1] TRUE
```

---

## Tabliczka logicznego dodawania i logicznego mnożenia

Przydatną do przedstawienia możliwych wyników operatora `|` jest tabliczka logicznego dodawania. Aby ją przedstawić zdefiniujemy wektor zawierający każdą z trzech możliwych wartości logicznych.

```
(TRUE_FALSE_NA <- c(TRUE, FALSE, NA))
```

```
## [1] TRUE FALSE NA
```

Funkcja `outer()` jako argumenty przyjmuje dwa wektory a następnie dla każdej pary wartości z pierwszego i drugiego wektora stosuje funkcję, która jest trzecim argumentem. Kolejne dwie linie dodają nazwy wierszy i kolumn, tak by na koniec wyprodukować ładnie wyglądającą tabliczkę logicznego dodawania. Jeżeli którykolwiek z argumentów ma wartość logicznej prawdy to wynikiem `lub` jest logiczna prawda.

```
tabliczka_dodawania <- outer(TRUE_FALSE_NA, TRUE_FALSE_NA, FUN = lub)
rownames(tabliczka_dodawania) <- TRUE_FALSE_NA
colnames(tabliczka_dodawania) <- TRUE_FALSE_NA
tabliczka_dodawania
```

##		TRUE	FALSE	<NA>
##	TRUE	TRUE	TRUE	TRUE
##	FALSE	TRUE	FALSE	NA
##	<NA>	TRUE	NA	NA

W podobny sposób możemy zbudować tabliczkę logicznego mnożenia, używając operatora `&`

```
tabliczka_mnozenia <- outer(TRUE_FALSE_NA, TRUE_FALSE_NA, FUN = `&`)
rownames(tabliczka_mnozenia) <- TRUE_FALSE_NA
colnames(tabliczka_mnozenia) <- TRUE_FALSE_NA
tabliczka_mnozenia
```

##		TRUE	FALSE	<NA>
##	TRUE	TRUE	FALSE	NA
##	FALSE	FALSE	FALSE	FALSE
##	<NA>	NA	FALSE	NA

Warto zwrócić uwagę na wynik (`TRUE | NA`). Wynikiem jest wartość `TRUE`, ponieważ bez znaczenia czy wartość nieokreślona okazałaby się prawdziwa czy fałszywa, jej logiczna suma z wartością `TRUE` dałoby wartość `TRUE`. Na podobnej zasadzie `FALSE & NA` zwraca wartość `FALSE`.

W programie R występują również dłuższe postacie operatorów `|` i `&` czyli `||` i `&&`. Pomiędzy formą dłuższą i krótszą występują dwie różnice. Forma krótsza pracuje na wektorach i wykonuje operacje element wektora po elemencie, podczas gdy forma dłuższa wykonuje operacje jedynie na pierwszych elementach wektorów i jako wynik zwraca jednoelementową wartość `TRUE` lub `FALSE`. Druga

różnica dotyczy zaawansowanych zastosowań, długa forma nie wykonuje ewaluacji prawego argumentu, jeżeli nie jest to niezbędne.

Przykładowo pierwsza linia poniższego przykładu wykona się poprawnie, ponieważ do określenia wyniku nie potrzebna jest ewaluacja funkcji `cat()`. Drugi przykład wykona funkcję `cat()` oraz zasygnalizuje błąd, ponieważ nie sposób wyniku funkcji `cat()` logicznie dodać do wartości `TRUE`.

```
TRUE || cat("Jestem tutaj !!!")
## [1] TRUE
TRUE | cat("Jestem tutaj !!!")
## Jestem tutaj !!!
## Error in TRUE | cat("Jestem tutaj !!!") :
##   operations are possible only for numeric,
```

---

## Wczytanie danych

Dane od których rozpoczniemy przykłady to `koty_ptaki` z pakietu `PogromcyDanych`. Aby te dane wczytać, wystarczy włączyć pakiet, instrukcja jak to zrobić znajduje się w odcinku 2.

Włączmy pakiet i użyjmy funkcji `head()` by wyświetlić pierwsze sześć wierszy.

```
library(PogromcyDanych)
head(koty_ptaki)
```

##	gatunek	waga	dlugosc	predkosc	habitat	zyw
## 1	Tygrys	300	2.5	60	Azja	
## 2	Lew	200	2.0	80	Afryka	
## 3	Jaguar	100	1.7	90	Ameryka	
## 4	Puma	80	1.7	70	Ameryka	
## 5	Leopard	70	1.4	85	Azja	
## 6	Gepard	60	1.4	115	Afryka	

---

## Statystyki opisowe

Na bazie zmiennej `druzyna` zbudujemy zmienną logiczną `czy_to_kot`, a na bazie zmiennej `waga` zbudujemy zmienną logiczną `czy_jest_ciezki`.

```
czy_to_kot <- koty_ptaki$druzyna == "Kot"
czy_jest_ciezki <- koty_ptaki$waga >= 10
```

Traktując zmienną logiczną jak dwuwartościową zmienną jakościową, możemy używać na niej np. funkcji do tabel liczebności, zarówno jednowymiarowych jak i dwuwymiarowych.

Patrząc na wyniki dla tabeli liczebności dla dwóch zmiennych jest tylko jeden zwierzak w tym zbiorze danych, który jest cięższy niż 10kg, ale nie jest kotem.



```
table(czy_to_kot)
```

```
## czy_to_kot  
## FALSE TRUE  
##      6      7
```

```
table(czy_jest_ciezki)
```

```
## czy_jest_ciezki  
## FALSE TRUE  
##      5      8
```

```
table(czy_to_kot, czy_jest_ciezki)
```

```
##              czy_jest_ciezki  
## czy_to_kot FALSE TRUE  
##      FALSE      5      1  
##      TRUE      0      7
```

---

## Operator negacji

Innym przydatnym operatorem jest logiczna negacja `!`. Zamienia ona wartość logicznej prawdy na fałsz i odwrotnie.

```
czy_to_kot
```

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE TI  
## [12] FALSE FALSE
```

```
!czy_to_kot
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FAI
```

```
## [12] TRUE TRUE
```

W operacjach arytmetycznych wartości logiczne TRUE są przekształcane na 1 a wartości FALSE na 0. Dzięki czemu, jeżeli chcemy policzyć ile wartości ma wartość TRUE to wystarczy wartość wektora zsumować. Poniższy przykład pokazuje ile jest kotów i nie kotów w zbiorze danych.

```
sum(czy_to_kot)
```

```
## [1] 7
```

```
sum(!czy_to_kot)
```

```
## [1] 6
```

Taki wektor wartości logicznych można mnożyć z wektorem wartości liczbowych. Poniższa instrukcja używa takiej operacji do policzenia sumy wagi wszystkich kotów w zbiorze danych. Mnożenie przez wektor wartości logicznych powoduje wyzerowanie określonych wartości (TRUE jest zamieniane na 1, FALSE na 0, a elementy obu wektorów są mnożone element po elemencie).

```
czy_to_kot * koty_ptaki$waga
```

```
## [1] 300 200 100 80 70 60 50 0 0 (
```

```
sum(czy_to_kot * koty_ptaki$waga)
```

```
## [1] 860
```

---

# Wczytanie danych

Zbiór danych `koty_ptaki` składa się z 13 wierszy. Można cały ten zbiór danych wyświetlić na ekranie. Nie zawsze potrzebujemy więc specjalnych statystyk opisowych by zrozumieć co się dzieje w takich małych zbiorach danych.

Dlatego dalsze ćwiczenia ze zmiennymi logicznymi przeprowadzimy na znacznie większym zbiorze danych `auta2012` z ponad 200 tysiącami wartości, który również znajduje się w pakiecie `PogromcyDanych`.

Opis tego zbioru danych znaleźć można w odcinku [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzanie\\_danych](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzanie_danych)

Wczytajmy ten zbiór danych i przyjrzyjmy się trzem pierwszym wierszom.

```
library(PogromcyDanych)
head(auta2012, 3)

## Source: local data frame [3 x 28]
##
##   Cena Waluta Cena.w.PLN Brutto.netto KM
## 1 49900     PLN   49900      brutto 140
## 2 88000     PLN   88000      brutto 156
## 3 86000     PLN   86000      brutto 150
## Variables not shown: Liczba.drzwi (fctr), Przebieg.w.km (dbl), Rodzaj.paliwa (fctr),
##   (fctr), Kraj.aktualnej.rejestracji (fctr)
```

```
## Pojazd.uszkodzony (fctr), Skrzynia.biegow
## Status.pojazdu.sprawadzonego (fctr), Wypos
## Rodzaj.paliwa.posortowany (fctr), Kolor_na
## Wyposazenie.dodatkowe_napis (chr), czy_me
## (lgl), szyby (lgl), MarkaModel (chr)
```

---

## Statystyki opisowe

Wykorzystajmy wartości logiczne oraz funkcję `table()`, aby sprawdzić ile aut w naszych danych jest zarejestrowanych w Polsce oraz ile z nich zostało wyprodukowanych przed 2007 rokiem. Tak jak w przypadku zmiennych jakościowych możemy wykorzystać funkcję `prop.table()` do wyznaczenia frakcji / procentów.

Co może być zaskakujące, mniej niż połowa ofert ma zadeklarowany kraj aktualnej rejestracji jako Polska.

```
pochodziZPolski <- auta2012$Kraj.aktualnej.rejestracji == "Polska"
table(pochodziZPolski)
```

```
## pochodziZPolski
##      FALSE      TRUE
## 116592    91010
```

```
prop.table(table(pochodziZPolski)) * 100
```

```
## pochodziZPolski
```

```
##      FALSE      TRUE
## 56.16131 43.83869
```

```
starszyNiz5Lat <- auta2012$Rok.produkcji < 200
table(starszyNiz5Lat)
```

```
## starszyNiz5Lat
##      FALSE      TRUE
## 66703 140899
```

```
prop.table(table(starszyNiz5Lat)) * 100
```

```
## starszyNiz5Lat
##      FALSE      TRUE
## 32.13023 67.86977
```

---

## Testy dla typów i wartości

Porównując wartości należy uważać na wartości nieokreślone, które nie zawsze zachowują się zgodnie z naszą intuicją, ale zawsze logicznie.

Przykładowo, operator `!=` testuje czy wartości są różne (to negacja operatora `==`). Jak sprawdzić czy wartości w wektorze są różne od wartości brakującej `NA`? Zaczniemy od przykładu jak tego nie robić.

```
czyOkreslonyPrzebieg <- auta2012$Przebieg.w.km
head(auta2012$Przebieg.w.km)
```

```
## [1] 41000 46500 8000 200000 169400 14110
```

```
head(czyOkreslonyPrzebieg)
```

```
## [1] NA NA NA NA NA NA
```

Wynikiem porównania `!= NA` jest wektor wartości `NA`. Jest tak ponieważ jeżeli prawa część porównania jest nieznana to też nie wiadomo czy wartość 41000 jest od niej różna czy nie.

Aby sprawdzić, czy dana wartość jest wartością brakującą możemy wykorzystać funkcję `is.na()`. Jako wynik zwraca `TRUE` jeżeli testowana wartość to `NA` i `FALSE` w przeciwnym przypadku.

Możemy użyć tej funkcji by policzyć procent wartości, dla których nie ma brakujących danych.

```
czyOkreslonyPrzebieg <- !is.na(auta2012$Przebieg)
head(czyOkreslonyPrzebieg)
```

```
## [1] TRUE TRUE TRUE TRUE TRUE TRUE
```

```
## a tutaj liczymy procent wartości
mean(czyOkreslonyPrzebieg) * 100
```

```
## [1] 80.94479
```

Funkcji do testowania jest więcej, zaczynają one swoje nazwy od `is.`, wystarczy więc wpisać te trzy znaki do konsoli i nacisnąć klawisz `TAB` by zobaczyć ich listę.

Część z nich sprawdza czy zmienna jest określonego typu, np. `is.factor()`, `is.numeric()`, część sprawdza czy zmienna ma określone wartości np. `is.na()` czy `is.nan()`.

---

## Logiczne ‘i’ oraz logiczne ‘lub’

Ponieważ operatory logiczne `lub` oraz `i` są często wykorzystywane przećwiczymy je jeszcze na zbiorze danych o samochodach. Zdefiniujemy dwa wektory wartości logicznych, weryfikujących czy samochód jest aktualnie zarejestrowany w Polsce oraz czy jest starszy niż pięć lat, a więc wyprodukowany przed rokiem 2007.

```
pochodziZPolski <- auta2012$Kraj.aktualnej.rejestracji == "Polska"
starszyNiz5Lat <- auta2012$Rok.produkcji < 2007
```

Teraz policzymy logiczną sumę oraz iloczyn obu wektorów. Następnie użyjemy funkcji `table` aby przedstawić wyniki.

```
pochodziZPolskiIScarszyNiz5Lat <- pochodziZPolski & starszyNiz5Lat
pochodziZPolskiLubStarszyNiz5Lat <- pochodziZPolski | starszyNiz5Lat
```

Tabela liczebności (tabela krzyżowa) dla obu zmiennych

```
table(pochodziZPolski, starszyNiz5Lat)
```

```
##                                starszyNiz5Lat
##  pochodziZPolski FALSE    TRUE
##                                FALSE  44025  72567
##                                TRUE   22678  68332
```

## Logiczne i (tak zwany logiczny iloczyn)

```
table(pochodziZPolskiIStarszyNiz5Lat)
```

```
##  pochodziZPolskiIStarszyNiz5Lat
##    FALSE    TRUE
##  139270   68332
```

## Logiczne lub (tak zwana logiczna suma)

```
table(pochodziZPolskiLubStarszyNiz5Lat)
```

```
##  pochodziZPolskiLubStarszyNiz5Lat
##    FALSE    TRUE
##    44025  163577
```

---

## Zadania:

- Sprawdź ile samochodów zarejestrowanych w Polsce ma cenę ofertową poniżej 2 000 pln.
- Sprawdź jaki procent samochodów ma silniki o pojemności ponad 1500 cm<sup>3</sup> oraz jest napędzanych olejem napędowym.



Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Napisy

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 14*

*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Co to znaczy: napisy](#)
- [Wczytanie danych](#)
- [Konwersja na napis](#)
- [Wyszukiwanie napisu](#)
- [Wyszukiwanie napisu](#)
- [Fragmentu napisów](#)
- [Zadanie](#)
- [Wczytanie danych](#)
- [Napisy](#)
- [Napisy](#)
- [Napisy](#)
- [Sklejanie napisów](#)
- [Sklejanie napisów](#)
- [Imiona dzieci](#)
- [Liczby, cechy jakościowe i napisy](#)
- [Zadania:](#)
- [Gdzie szukać dodatkowych informacji](#)

# O czym jest ten odcinek

Analizując dane spotkamy się z różnymi rodzajami zmiennych. Bardzo często są to napisy lub zmienne, które chcemy do napisów przekształcić (np. data, którą można przekształcić na napisy by wyłuskać rok i miesiąc).

W tym odcinku nauczymy się:

- jak przekształcać zmienne z napisów i na napisy,
- jakie podstawowe operacje można wykonywać na napisach,
- jak podsumowywać / opisywać napisy.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012`, oba dostępne w pakiecie `PogromcyDanych`.

---

## Co to znaczy: napisy

Osoby mające doświadczenie w pracy z takimi językami jak C++ czy Java są przyzwyczajone do podziału typów na znaki, napisy (łańcuchy znaków) i wektory napisów. W programie R nie ma tego podziału, są tylko wektory napisów.

Wektor może być jednoelementowy i długość tego jednego elementu może wynosić jeden, ale wciąż to wektor napisów. Takie wektory oznaczane są klasą `character`.

```
class("A")
```

```
## [1] "character"
```

Napisy spotkać można w różnych kontekstach. Najbardziej naturalnym jest taki, że w zbiorze danych były zebrane wypowiedzi. Np. pobierając dane z Twittera, jedną ze zmiennych opisujących pojedyncze „ćwierknięcie” jest jego treść, czyli napis o długości do 140 znaków.

Napisy pojawiają się też, gdy wczytywane są dane ilościowe lub jakościowe, ale z jakiegoś powodu w procesie wczytywania dane te zinterpretowane zostały jako napisy (np. przez źle określone formatowanie). W takim przypadku napisy przekształcamy często na pożądaný typ, np. ilościowy, jakościowy, datę itp.

```
as.numeric("2012")      # konwersja na liczbę
```

```
## [1] 2012
```

```
as.factor(c("A", "B", "A", "A")) # konwersja na
```

```
## [1] A B A A
```

```
## Levels: A B
```

```
as.Date("2012-01-01")    # konwersja na datę
```

# Wczytanie danych

Dane od których rozpoczniemy przykłady to `koty_ptaki` z pakietu `PogromcyDanych`. Aby te dane wczytać, wystarczy włączyć pakiet - instrukcja jak to zrobić znajduje się w odcinku 2.

Włączmy pakiet i użyjmy funkcji `head()`, by wyświetlić pierwsze sześć wierszy.

```
library(PogromcyDanych)
head(koty_ptaki)
```

##	gatunek	waga	dlugosc	predkosc	habitat	zywe
## 1	Tygrys	300	2.5	60	Azja	
## 2	Lew	200	2.0	80	Afryka	
## 3	Jaguar	100	1.7	90	Ameryka	
## 4	Puma	80	1.7	70	Ameryka	
## 5	Leopard	70	1.4	85	Azja	
## 6	Gepard	60	1.4	115	Afryka	

Każdy wiersz opisuje jeden gatunek. Jak wiemy z poprzedniego odcinka, zmienne `gatunek`, `habitat` i `druzyna` to zmienne jakościowe. Przekształcimy je na napisy, aby zademonstrować podstawowe operacje na napisach.

# Konwersja na napis

Wybermy kolumnę `habitat`. W zbiorze danych `koty_ptaki` jest to zmienna jakościowa. Użyjemy funkcji `as.character()` by przekształcić ją na napisy. Otrzymane napisy zapiszemy w kolumnie `habitat_napis`.

```
koty_ptaki$habitat

##      [1] "Azja"      "Afryka"    "Ameryka"   "Ameryka"
##      [7] "Azja"      "Euroazja"  "Afryka"    "Północna"
##     [13] "Północna"
```

```
koty_ptaki$habitat_napis <- as.character(koty_ptaki$habitat)
koty_ptaki$habitat_napis

##      [1] "Azja"      "Afryka"    "Ameryka"   "Ameryka"
##      [7] "Azja"      "Euroazja"  "Afryka"    "Północna"
##     [13] "Północna"
```

Funkcja `length()` dla każdego wektora sprawdza jego długość (liczbę napisów). Funkcja `nchar()` dla napisu sprawdza liczbę znaków w każdym napisie w wektorze (długość każdego napisu).

```
length(koty_ptaki$habitat_napis)

##      [1] 13

nchar(koty_ptaki$habitat_napis)

##      [1] 4 6 7 7 4 6 4 8 6 6 6 6 8
```

# Wyszukiwanie napisu

Częstą operacją na napisach jest wyszukanie tych, które pasują do wzorca. Jeżeli chodzi nam o zgodność co do znaku, to wygodne będzie użycie funkcji `which()`. Jako wynik zwraca indeksy, dla których określony warunek jest prawdziwy.

Które wartości wektora `habitat_napis` to `Azja`?

```
which(koty_ptaki$habitat_napis == "Azja")
```

```
## [1] 1 5 7
```

Które wartości wektora `habitat_napis` to `Azja` lub `Afryka`?

```
which(koty_ptaki$habitat_napis %in% c("Azja", "Afryka"))
```

```
## [1] 1 2 5 6 7 9
```

Często jednak, zamiast wymieniać wszystkie wartości, które chcemy wyszukać, wygodniejsze jest opisanie tych wartości przez pewien wzorzec. Do takiego wyszukiwania służy funkcja `grep()`. Testuje ona, w którym z napisów występuje określony ciąg znaków, a jako wynik zwraca numery napisów z tym ciągiem.

Sprawdźmy, które wiersze w kolumnie `habitat_napis` posiadają w nazwie znak `A`. Pierwszym argumentem funkcji `grep()` jest wzorzec, drugim wektor napisów. Możemy te argumenty podać w innej kolejności, ale wtedy zmuszeni jesteśmy do podawania ich nazwy.

```
grep("A", koty_ptaki$habitat_napis)
```

```
## [1] 1 2 3 4 5 6 7 9
```

Równoważnie

```
grep(koty_ptaki$habitat_napis, pattern="A")
```

```
## [1] 1 2 3 4 5 6 7 9
```

---

## Wyszukiwanie napisu

Jeżeli zamiast indeksów chcemy otrzymać wartości, które zostały dopasowane, wtedy powinniśmy ustawić argument `value` na wartość `TRUE`.

```
grep("A", koty_ptaki$habitat_napis, value = TRUE)
```

```
## [1] "Azja"      "Afryka"    "Ameryka"   "Ameryka"  
## [8] "Afryka"
```

Innym przydatnym argumentem funkcji `grep()` jest możliwość określenia, że wyszukiwanie ma być wykonane



z pominięciem informacji o wielkości znaków. Służy do tego argument `ignore.case=TRUE`.

```
grep("A", koty_ptaki$habitat_napis, ignore.case=TRUE)
## [1] 1 2 3 4 5 6 7 8 9
```

Określając wzorzec możemy wykorzystać wyrażenia regularne. Pozwalają one opisać pewną prawidłowość, którą chcemy wyszukać w danych. Wyrażenia regularne są często stosowane, aby sprawdzić czy napis jest na przykład mailem, numerem telefonu lub kodem pocztowym.

Więcej o wyrażeniach regularnych można przeczytać choćby na Wikipedii [http://pl.wikipedia.org/wiki/Wyra%C5%BCenie\\_regularne](http://pl.wikipedia.org/wiki/Wyra%C5%BCenie_regularne)

Poniższe wyrażenie, sprawdza czy początek napisu to litera A lub E.

```
grep("^[AE]", koty_ptaki$habitat_napis)
## [1] 1 2 3 4 5 6 7 8 9
```

---

## Fragmentu napisów

Kolejną przydatną funkcją do operowania na napisach jest

`substr()`. Pozwala ona z napisu wyciąć fragment o określonej pozycji, drugim i trzecim argumentem są indeksy początku i końca napisu, który chcemy wyciąć.

Przykładowo, w ten sposób możemy z dat wycinać informację o latach lub miesiącach.

```
daty <- c("2014-01-01", "2015-03-15", "2010-12-31")
substr(daty, 1, 4)
```

```
## [1] "2014" "2015" "2010"
```

```
substr(daty, 6, 7)
```

```
## [1] "01" "03" "12"
```

Innym sposobem wyluskiwania elementu napisu jest podanie wzorca który rozdziela istotne elementy napisu, np. słowa. W poniższym przykładzie dwa zdania są rozbijane na wyrazy (a dokładniej rozbijane na fragmenty rozdzielane spacją). Wynikiem jest dwuelementowa lista wektorów. Wybieramy pierwszy wektor i odczytujemy z niego drugie słowo.

```
zdanie <- c("W Szczecinie chrząszcz brzmi w trzcinie")
(podzielony <- strsplit(zdanie, " "))
```

```
## [[1]]
```

```
## [1] "W" "Szczecinie" "chrząszcz" "brzmi" "w" "trzcinie"
```

```
## [5] "w" "trzcinie"
```

```
##
```

```
## [[2]]
```

```
## [1] "Zab" "zupa" "zębowa" "dąb" "zu
```

```
slova1 <- podzielony[[1]]  
slova1[2]
```

```
## [1] "Szczepreszynie"
```

---

## Zadanie

- Wyznacz indeksy wierszy w zbiorze danych `koty_ptaki`, w których `gatunek` zawiera dużą lub małą literę `s`.
  - Wyznacz wszystkie wiersze, w których opisano jakiś gatunek sokoła.
- 

## Wczytanie danych

Zbiór danych `koty_ptaki` składa się z 13 wierszy. Można cały ten zbiór danych wyświetlić na ekranie. Nie zawsze potrzebujemy więc specjalnych statystyk opisowych, by zrozumieć co się dzieje w takich małych zbiorach danych.

Dlatego dalsze ćwiczenia z napisami przeprowadzimy na znacznie większym zbiorze danych `auta2012`, z ponad 200 tysiącami wartości, który również znajduje się w

pakiecie PogromcyDanych.

Opis tego zbioru danych znaleźć można w odcinku [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

Wczytajmy ten zbiór danych i przyjrzyjmy się dwóm pierwszym wierszom.

```
library(PogromcyDanych)
head(auta2012, 2)

## Source: local data frame [2 x 28]
##
##      Cena Waluta Cena.w.PLN Brutto.netto KM
## 1 49900     PLN      49900      brutto 140
## 2 88000     PLN      88000      brutto 156
## Variables not shown: Liczba.drzwi (fctr), Przebieg.w.km (dbl), Rodzaj.paliwa (fctr),
##      (fctr), Kraj.aktualnej.rejestracji (fctr), Pojazd.uszkodzony (fctr), Skrzynia.biegow
##      Status.pojazdu.sprowadzonego (fctr), Wypos Rodzaj.paliwa.posortowany (fctr), Kolor_na
##      Wyposazenie.dodatkowe_napis (chr), czy_me (lgl), szyby (lgl), MarkaModel (chr)
```

Dwie kolumny Wyposazenie.dodatkowe oraz Kolor będą nas szczególnie interesowały, jeżeli chodzi o analizę napisów.

# Napisy

Wczytując dane do programu R większość funkcji, domyślnie zamienia napisy na zmienne jakościowe. To zagadnienie było szerzej omawiane w odcinku 5.

W sytuacjach, gdy chcemy przetwarzać napisy, musimy zmienne jakościowe przekształcić na napisy używając funkcji `as.character()`.

```
auta2012$Kolor_napis <- as.character(auta2012$Kolor)
auta2012$Wyposazenie.dodatkowe_napis <- as.character(auta2012$Wyposazenie.dodatkowe)
```

---

## Napisy

W analizie danych często stoimy przed potrzebą stworzenia nowych cech na bazie starych. Przykładowo, jeżeli spojrzymy na wektor kolorów samochodów, możemy uznać że interesującą cechą byłoby określenie, czy lakier jest koloru metalicznego czy nie.

```
sort(table(auta2012$Kolor_napis))
```

##		
##	rozowy	rozowy-metallic
##	15	62
##	brezowy	wisniowy
##	141	193
##	zloty	zolty-metallic
##	219	231
##	pomaralczowy-metallic	grafitowy

##	459	542
##	bordowy	zolty
##	742	960
##	fioletowy-metallic	bialy-metallic
##	1101	1542
##	brezowy-metallic	zielony
##	1988	2027
##	srebrny	zloty-metallic
##	2553	3122
##	niebieski	granatowy
##	4037	4057
##	czerwony	grafitowy-metallic
##	7465	9481
##	zielony-metallic	granatowy-metallic
##	10148	10756
##	niebieski-metallic	
##	12584	25997
##	srebrny-metallic	
##	40626	

Można zrobić to na kilka sposobów. Najłatwiejszym jest użycie funkcji `grepl()`, o takich samych parametrach jak funkcja `grep()` z tym wyjątkiem, że wynikiem jest wektor wartości logicznych TRUE/FALSE określający czy dany wzorzec został odnaleziony czy nie w napisach.

```
auta2012$czy_metallic <- grepl("metallic", auta2012$nazwa)
table(auta2012$czy_metallic)
```

```
##
## FALSE TRUE
## 66202 141400
```

# Napisy

W podobny sposób, można stworzyć dodatkowe kolumny określające, czy auto ma na wyposażeniu klimatyzację lub czy szyby są przyciemniane.

W drugim przykładzie warto zwrócić uwagę na to, że argumenty są podane w kolejności innej niż domyślna. Dlatego argument ze wzorcem musiał mieć dodatkowo wskazaną nazwę `pattern=`.

```
auta2012$maKlimatyzacje <- grepl("klimatyzacja",
table(auta2012$maKlimatyzacje))
```

```
##
##  FALSE    TRUE
##  44642 162960
```

Około 20% oferowanych aut ma przyciemniane szyby.

```
auta2012$szyby <- grepl(x=auta2012$Wyposazenie,
                        pattern="przycie",
table(auta2012$szyby))
```

```
##
##  FALSE    TRUE
## 165480  42122
```

---

## Sklejanie napisów

Operacją przeciwną do dzielenia napisu na fragmenty jest sklejanie kilku napisów w jeden.

Do tego celu można wykorzystać funkcję `paste()` lub `paste0()`.

Przedstawmy jej działanie na prostym przykładzie. Będziemy sklejać wektor dziesięciu liter, dziesięciu cyfr i jednego napisu.

Funkcje w programie R, które operują na wektorach (większość), jeżeli mają dwa argumenty o różnej długości to krótszy argument jest zwielokrotniony tylekrotnie, aby dorównał długością dłuższemu (tzw. recycling rule). Z tego powodu trzeci argument - pojedyncza kropka - będzie zwielokrotniona i sklejona z każdym z dziesięciu elementów wektorów `litery` i `cyfry`.

```
litery <- LETTERS[1:10]
cyfry  <- 1:10
```

Funkcja `paste()` skleja wartości rozdzielając je separatorem, którym domyślnie jest spacja. Znak separatora można zmienić wskazując argument `sep`. Funkcja `paste0()` tym różni się od funkcji `paste()`, że separator jest pustym napisem.

```
paste(litery, cyfry, ".")
```



```
## [1] "A 1 ." "B 2 ." "C 3 ." "D 4 ." "E  
## [8] "H 8 ." "I 9 ." "J 10 ."
```

```
paste(litery, cyfry, ".", sep="-")
```

```
## [1] "A-1-." "B-2-." "C-3-." "D-4-." "E-  
## [8] "H-8-." "I-9-." "J-10-."
```

```
paste0(litery, cyfry, ".")
```

```
## [1] "A1." "B2." "C3." "D4." "E5." "F6"
```

---

## Sklejanie napisów

Przedstawmy funkcję do sklejania napisów na przykładzie sklejania nazwy marki i modelu dla auta.

W pierwszej linii skleimy oba wektory, używając za separator `:`. Następnie zliczymy ile było wystąpień takich par i uporządkujemy je w kolejności malejącej.

Wyświetlamy 25 najczęstszych modeli.

```
auta2012$MarkaModel <- paste(auta2012$Marka,  
statystykiMarkiModelu <- sort(table(auta2012$Ma  
head(statystykiMarkiModelu, 25)
```

```
##  
## Volkswagen: Passat Opel: Astra Voll
```

##	6883	6348	
##	Ford: Focus	Audi: A4	
##	5691	4280	
##	Skoda: Octavia	Opel: Vectra	Ren
##	3954	3815	
##	Renault: Scenic	Renault: Laguna	
##	3546	3288	
##	Opel: Corsa	Volkswagen: Polo	
##	3029	2714	
##	Renault: Clio	Opel: Zafira	
##	2469	2040	
##	Ford: Fiesta	Peugeot: 206	
##	1879	1863	
##	BMW: 320	Honda: Civic	
##	1791	1703	
##	Toyota: Yaris		
##	1552		

---

## Imiona dzieci

W pakiecie `PogromcyDanych` jest również dostępny zbiór danych `imiona_warszawa` ze statystykami popularności imion nadawanych noworodkom w Warszawie w różnych okresach czasu.

Używając tego zbioru danych, możemy przetestować regułę mówiącą, że imię dziewczynki kończy się na literkę ‘a’, a chłopca nie. Jakiś czas temu za jedyny wyjątek podawano męskie imię Bonawentura. Czy to się zmieniło?

Wybermy do analizy tylko imiona chłopców (indeksowanie `imiona_warszawa$plec == "M"`). Ponieważ każde imię powtarza się dla każdego miesiąca, to funkcją `unique()` usuniemy wszystkie duplikaty.

Następnie funkcją `grep()` wybierzemy wszystkie imiona, których ostatnia litera to `a` (wyrażenie regularne `a$`) i wyświetlimy te imiona. Bonawentury nie widać ale są inne imiona.

```
imiona_chlopcow <- imiona_warszawa[imiona_warszawa$plec == "M", ]
same_imiona     <- unique(imiona_chlopcow$imie)
grep("a$", same_imiona, value = TRUE)
```

```
## [1] "Kosma" "Nikita"
```

Podobne operacje powtórzymy dla dziewcząt. Tym razem do wyrażenia regularnego podamy wzorzec wszystkie litery poza `a`, czyli `[^a]$`. Takich dziewczęcych imion jest wiele.

```
imiona_dziewczat <- imiona_warszawa[imiona_warszawa$plec == "F", ]
same_imiona     <- unique(imiona_dziewczat$imie)
grep("[^a]$", same_imiona, value = TRUE)
```

```
## [1] "Abigail" "Berenike" "Carmen" "Chloe"
## [7] "Inez" "Ingrid" "Karin" "Lili"
## [13] "Miriam" "Naomi" "Nel" "Nelly"
## [19] "Nikol" "Nikole" "Noemi" "Rache"
```

---

# Liczby, cechy jakościowe i napisy

Funkcja `as.character()` może być przydatna jeszcze w jednej sytuacji, tzn. gdy zmienną jakościową chcemy przekształcić na zmienną liczbową. Nie powinniśmy tego zrobić bezpośrednio, ale poprzez funkcję `as.character()`.

Pokażmy ten problem na przykładzie. Zamienimy zmienne na zmienne jakościowe funkcją `as.factor()` a następnie na liczby funkcją `as.numeric()`. Okazuje się jednak, że wynik nie jest zgodny z oczekiwaniami.

```
x <- c(2, 4, 5.5)
(fx <- as.factor(x))
```

```
## [1] 2    4    5.5
## Levels: 2 4 5.5
```

Uwaga! Program R nie wie że to są napisy, a zmienną jakościową zamienia na kolejne liczby całkowite.

```
as.numeric(fx)
```

```
## [1] 1 2 3
```

Zmienne jakościowe przez funkcję `as.numeric()` zamieniane są na kolejne liczby całkowite. Jeżeli chcemy odzyskać liczby, które są treścią czynników, to musimy je wpierw zamienić na napisy, funkcją `as.character()`.

```
as.numeric(as.character(fx))
```

```
## [1] 2.0 4.0 5.5
```

Alternatywnie, można wykorzystać odrobinę wydajniejszy i również odrobinę mniej czytelny sposób zamiany zmiennych jakościowych na liczby.

```
as.numeric(levels(fx))[fx]
```

```
## [1] 2.0 4.0 5.5
```

---

## Zadania:

- Sprawdź ile samochodów ma zainstalowany autoalarm (oznaczony jako ‘autoalarm’)
- Sprawdź ile samochodów ma lakier w metalicznym kolorze (oznaczony jako ‘metallic’)

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

---

## Gdzie szukać dodatkowych informacji

Jedynie zarysowaliśmy temat analizy napisów.

Przy przetwarzaniu większych korpusów tekstu, pojawia się wiele interesujących tematów, takich jak analiza znaczenia, analiza częstościowa, tagowanie itp.

- Więcej o podstawowych operacjach do pracy z napisami przeczytać można w rozdziale 2.1 „Przewodnika po pakiecie R” GiS 2013. Rozdział ten jest dostępny bezpłatnie na stronie <http://biecek.pl/R/>
- Gdy pracujemy z dużymi korpusami, możemy przyspieszyć obliczenia stosując pakiet `stringr`. Bardzo dobrze przygotowany materiał opisujący funkcje tego pakietu jest udostępniony przez Gastona Sancheza na stronie [http://gastonsanchez.com/Handling\\_and\\_Processing\\_5](http://gastonsanchez.com/Handling_and_Processing_5)
- Gdy pracujemy z napisami bardzo użyteczne są wyrażenia i metaznaki. Krótkie ale treściwe omówienie wyrażeń regularnych, opracowane przez S. Jonesa znajduje się w dokumencie „String manipulation in R” [http://www3.nd.edu/~sjones20/JonesUND/BioStats\\_16-13.pdf](http://www3.nd.edu/~sjones20/JonesUND/BioStats_16-13.pdf).
- Interesującą alternatywą wspierającą bardziej egzotyczne języki (bardziej niż angielski) jest pakiet `stringi`. Można o nim posłuchać na wystąpieniu

Marka Gągolewskiego z II Spotkania Entuzjastów R.

Materiał wideo dostępny jest na stronie

<http://smarterpoland.pl/SER/#SERII>

# Daty

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 15*

*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Daty i czas](#)
- [Daty](#)
- [Czas calendar time](#)
- [Czas local time](#)
- [Pakiet lubridate](#)
- [Pakiet lubridate](#)
- [Zadania](#)
- [Więcej informacji](#)

## O czym jest ten odcinek

Analizując dane spotkamy się z różnymi rodzajami zmiennych. Często spotykanym typem okazują się zmienne związane z datami i czasem.

W tym odcinku nauczymy się:



- jak tworzyć w programie R obiekty opisujące daty i czasy,
- jakie podstawowe operacje można wykonywać na datach i czasie,
- jak podsumowywać / opisywać daty.

Do ilustracji tych zagadnień wykorzystamy zbiory danych tworzone ad hoc.

---

## Daty i czas

W programie R istnieją trzy podstawowe typy zmiennych opisujące daty lub czas.

- Klasa `Date` służy do opisywania dat z dokładnością do dnia. Na datach można wykonywać operacje takie jak odejmowanie dwóch dat, dodawanie liczby do dnia itp.
- Klasa `POSIXct`, służy do opisywania czasu z dokładnością do sekundy. Czas jest pamiętany jako liczba sekund od określonego początku. Sufiks `ct` oznacza *calendar time*.
- Klasa `POSIXlt`, służy do opisywania czasu w formie listy wartości. Sufiks `lt` oznacza *local*

*time.*

---

# Daty

Konstrukтором klasy `Date` jest funkcja `as.Date()`.

Jako pierwszy argument przyjmuje wektor napisów opisujących daty. Drugi opcjonalny argument określa formatowanie daty. Domyślne formatowanie to rok-miesiąc-dzień.

```
as.Date("2015-02-22")
```

```
## [1] "2015-02-22"
```

```
as.Date("02/22/2015", format = "%m/%d/%Y")
```

```
## [1] "2015-02-22"
```

```
as.Date("February 2, 2015", format = "%B %d, %Y")
```

```
## [1] "2015-02-02"
```

Aby uzyskać dokładną pomoc dotyczącą oznaczeń w formatowaniu daty należy otworzyć plik pomocy instrukcją `?strptime`.

Obiekty klasy `Date` można tworzyć także na podstawie liczb całkowitych lub obiektów klasy `POSIXct`, w obu

przypadkach przy pomocy funkcji `as.Date()`.

---

## Czas calendar time

Konstrukтором klasy `POSIXct` jest funkcja `as.POSIXct()`.

Jako pierwszy argument przyjmuje wektor napisów opisujących chwile czasu. Drugi opcjonalny argument określa formatowanie daty. Domyślne formatowanie to rok-miesiąc-dzień godzina:minuta:sekunda.

Aby uzyskać dokładną pomoc dotyczącą oznaczeń w formatowaniu daty należy otworzyć plik pomocy instrukcją `?strptime`.

```
(czas1 <- as.POSIXct("2015-02-13 12:56:26"))
```

```
## [1] "2015-02-13 12:56:26 CET"
```

```
(czas2 <- as.POSIXct("14022015 12:56:26", format = "%d%m%Y %H:%M:%S"))
```

```
## [1] "2015-02-14 12:56:26 CET"
```

Na czasach można wykonywać takie operacje jak odejmowanie czy dodawanie do określonego przedziału czasu (dodanie liczby całkowitej, domyślnie dodaje określoną liczbę sekund).

```
## różnica czasów
```

```
czas2 - czas1
```

```
## Time difference of 1 days
```

```
czas1
```

```
## [1] "2015-02-13 12:56:26 CET"
```

```
czas1 + 30
```

```
## [1] "2015-02-13 12:56:56 CET"
```

Funkcja `Sys.time()` jako wynik zwraca aktualny czas w formacie `POSIXct`.

```
Sys.time()
```

```
## [1] "2015-05-01 18:45:56 CEST"
```

```
Sys.time() - czas1
```

```
## Time difference of 77.20105 days
```

---

## Czas local time

Konstruktorem klasy `POSIXlt` jest funkcja `as.POSIXlt()`.

Jako pierwszy argument przyjmuje wektor napisów opisujących chwile czasu. Drugi opcjonalny argument określa formatowanie daty. Opis formatowania stosuje się taki sam, jak w przypadku klasy `POSIXct`.

```
(czas1 <- as.POSIXlt("2015-02-13 12:56:26"))
```

```
## [1] "2015-02-13 12:56:26 CET"
```

```
(czas2 <- as.POSIXlt("14022015 12:56:26", format = "%d%b%Y %H:%M:%S"))
```

```
## [1] "2015-02-14 12:56:26 CET"
```

Obiekty klasy `POSIXlt` to listy, można się do ich elementów odwoływać przez nazwę.

```
czas1$sec
```

```
## [1] 26
```

```
czas1$min
```

```
## [1] 56
```

```
czas1$zone
```

```
## [1] "CET"
```

Wiele funkcji można stosować zamiennie do obiektów klasy `POSIXlt` i `POSIXct`

```
## różnica czasów  
czas2 - czas1
```

```
## Time difference of 1 days
```

```
czas1 + 30
```

```
## [1] "2015-02-13 12:56:56 CET"
```

# Pakiet lubridate

W podstawowym programie R operacje na datach i czasie nie zawsze są proste. Tzn. proste operacje są proste, ale złożone już niekoniecznie.

Pakiet `lubridate` zawiera zestaw funkcji ułatwiający pracę z datami.

Jedną z takich funkcji jest `now()`, która zwraca obiekt opisujący bieżącą chwilę.

```
library(lubridate)
now()
```

```
## [1] "2015-05-01 18:45:56 CEST"
```

```
today()
```

```
## [1] "2015-05-01"
```

Uproszczono nazwy funkcji, służące do konwertowania napisów na czas. Nazwa funkcji staje się opisem formatu.

```
ymd_hms("2015-02-14 23:59:59")
```

```
## [1] "2015-02-14 23:59:59 UTC"
```

```
(czas3 <- mdy_hm("02/14/15 08:32"))
```

```
## [1] "2015-02-14 08:32:00 UTC"
```

# Pakiet lubridate

Z dat można wyciągać takie informacje jak dzień tygodnia, dzień miesiąca itd. Funkcją `wday()` można odczytać dzień tygodnia, podobnie funkcje `day()`, `month()` i `year()` opisują składowe daty.

```
wday(czas3, label = TRUE)
```

```
## [1] Sat  
## Levels: Sun < Mon < Tues < Wed < Thurs < Fri
```

Tydzień w roku, dzień miesiąca, miesiąc i rok.

```
week(czas1)
```

```
## [1] 7
```

```
day(czas1)
```

```
## [1] 13
```

```
month(czas1)
```

```
## [1] 2
```

```
year(czas1)
```

```
## [1] 2015
```

Na czasach można wykonywać wygodne operacje używając funkcji `days()`, `months()`, `years()`,

```
minutes(), seconds()).
```

```
czas3 + hours(4)
```

```
## [1] "2015-02-14 12:32:00 UTC"
```

```
czas3 + days(2) + months(4) + years(1)
```

```
## [1] "2016-06-16 08:32:00 UTC"
```

---

## Zadania

- Używając odpowiedniego formatowania zamień napis `01-15-2015 10:20:59` na obiekt klasy `POSIXct`.
  - Oblicz liczbę dni pomiędzy 1 września 1945 roku a 8 maja 1945.
  - Sprawdź jaki dzień tygodnia będzie za 100 dni od dziś
- 

## Więcej informacji

Przydatne wskazówki jak pracować z czasem i datami znaleźć można w artykule *Handling date-times in R* Cole Beck



<http://biostat.mc.vanderbilt.edu/wiki/pub/Main/ColeBeck/>

Szczegółowy opis pakietu `lubridate` znajduje się w artykule z JSS <http://www.jstatsoft.org/v40/i03/paper>

Wiele interesujących informacji o pracy z czasem i datami znajduje się tutaj:

[http://en.wikibooks.org/wiki/R\\_Programming/Times\\_and\\_](http://en.wikibooks.org/wiki/R_Programming/Times_and_)

# Czyszczenie i wprowadzenie do obróbki danych

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 16*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Tworzenie danych przez funkcję data.frame\(\)](#)
- [Zamiana nazwy kolumny](#)
- [Zamiana przecinka na kropkę](#)
- [Normalizacja kolumny lub wiersza](#)
- [Usuwanie lub zastępowanie brakujących danych](#)
- [Nazwy czynników w zmiennej jakościowej](#)
- [Podmiana dowolnej wartości](#)
- [Zadania](#)
- [Co dalej](#)

**O czym jest ten odcinek**

Praca z danymi to w dużej części czyszczenie i wielokrotna zmiana reprezentacji. Po co dane czyścić? Zarówno dane zbierane przez ludzi, jak i przez automatyczne czujniki, mają tendencje do zawierania braków, obserwacji odstających lub zwykłych błędów. Aby móc sensownie analizować dane, musimy przekształcić je do postaci w której wyniki analiz będą wiarygodne.

Czyszczenie to bardzo istotna faza. Bez względu bowiem jak zaawansowana i dobrze dobrana jest nasza technika analizy danych, jeżeli na wejściu będą śmieciowe dane to na wyjściu będą śmieciowe wyniki. *Garbage in garbage out*

W tym odcinku nauczymy się:

- jak edytować zbiory danych by doprowadzić je do spójnej postaci,
- jak zastępować jedne wartości przez inne,
- jak identyfikować wartości odstające / zbędne.

Mając dane wyczyszczone, możemy przejść do kolejnych etapów, czyli wstępnej obróbki, analizy i wizualizacji.

---

## Tworzenie danych przez funkcję

# data.frame()

Na potrzeby ćwiczeń z czyszczenia danych stworzymy sztuczną ramkę danych z kilkoma celowo dodanymi błędami.

Ramkę danych stworzymy funkcją `data.frame()`. Kolejne argumenty to wektory, które zostaną zamienione na kolejne kolumny w ramce danych. Ostatni argument `stringsAsFactors = FALSE` określa, że napisy mają NIE być zamieniane na zmienne jakościowe (domyślne dochodzi do zamiany).

```
DF <- data.frame(
  imie = c("Maja", "Anna", "Zosia", "Anna"),
  wiek = c("40", "12,5", "25", "16.6"),
  numer = c(1, 2, NA, 4),
  oculo = factor(c("niebieskie", "jasno-n:
stringsAsFactors = FALSE)
```

DF

##		imie	wiek	numer	oczo
## 1	Maja	40	1	niebieskie	
## 2	Anna	12,5	2	jasno-niebieskie	
## 3	Zosia	25	NA	ciemne	
## 4	Anna	16.6	4	niebieskie	

W wynikowej ramce będziemy chcieli poprawić następujące błędy:

- nazwę kolumny `oczo` zamienić na `oczy`,
- zmienną `wiek` zamienić na liczbową, w tym celu trzeba w pierw zamienić `,` na `.`,
- imię `Anna` zamienić na `Joanna`,
- kolory oczu zamienić na dwie grupy: `niebieskie` i `ciemne`.

## Zamiana nazwy kolumny

Aby wyświetlić nazwy kolumn można wykorzystać funkcję `colnames()`. Wynikiem jest wektor napisów, można na elementach tego wektora używać indeksów podobnie jak w poniższym przykładzie. Analogicznie funkcja `rownames()` pozwala na pracę z nazwami wierszy.

```
colnames(DF)
```

```
## [1] "imie" "wiek" "numer" "oczo"
```

```
## czwarty element wektora
colnames(DF)[4]
```

```
## [1] "oczo"
```

Funkcji `colnames()` można również użyć by zmienić nazwy kolumn. Poniższe dwie instrukcje wywołają funkcję `colnames<-()`, która zmienia nazwy kolumn. Można jednocześnie zmienić wszystkie lub tylko wybrane elementy wektora nazw kolumn.

Obie poniższe instrukcje poprawią nazwę ostatniej kolumny.

```
colnames(DF) <- c("imie", "wiek", "numer", "oczy")
colnames(DF)[4] <- "oczy"
```

W wyniku mamy więc zbiór danych

DF					
##		imie	wiek	numer	oczy
##	1	Maja	40	1	niebieskie
##	2	Anna	12,5	2	jasno-niebieskie
##	3	Zosia	25	NA	ciemne
##	4	Anna	16.6	4	niebieskie

## Zamiana przecinka na kropkę

Jeżeli pracujemy na danych wprowadzanych przez człowieka to często zdarza się, że raz wpisuje on jako separator dziesiętny znak . (kropka), a raz , (przecinek). Jeszcze więcej problemów tego typu występuje, gdy dane wprowadza kilka osób.

```
DF$wiek
```

```
## [1] "40" "12,5" "25" "16.6"
```

Zamienimy przecinek na kropkę i zamienimy napis na liczbę.

Do podmiany znaków użyjemy funkcji `gsub()`, (opisanej także w odcinku 14 tego sezonu). Nazwy argumentów `pattern` i `replacement` można by pominąć, ponieważ są w domyślnej kolejności, ale wpisujemy je tutaj dla czytelności.

Wynik przekazujemy do funkcji `as.numeric()`, która zamienia napisy na liczby. Ostatecznie wynik przypisujemy do kolumny `wiek`.

Dwie linie opatrzyliśmy nawiasami `()` aby wynik przypisania został wyświetlony na ekranie.

```
(tmp <- gsub(pattern = ",", replacement = ".",
```

```
## [1] "40"      "12.5"    "25"      "16.6"
```

```
(DF$wiek <- as.numeric(tmp))
```

```
## [1] 40.0 12.5 25.0 16.6
```

W wyniku mamy więc poprawiony zbiór danych

```
DF
```

##	imie	wiek	numer	oczy
## 1	Maja	40.0	1	niebieskie
## 2	Anna	12.5	2	jasno-niebieskie
## 3	Zosia	25.0	NA	ciemne
## 4	Anna	16.6	4	niebieskie

# Normalizacja kolumny lub wiersza

W analizie danych często spotykamy się z sytuacją, gdy dane w kolumnie chcemy unormować.

Unormować, czyli wycentrować, a więc usunąć z nich wartość średnią (tak by średnia była równa 0), i przeskalować, czyli podzielić tak, aby odchylenie standardowe było równe 1.

Do normalizacji można wykorzystać funkcję `scale()`. Domyślnie centruje i skaluje ona dane, ale ustawiając dodatkowe argumenty możemy wyłącznie wycentrować lub przeskalować dane.

Wynik funkcji `scale()` możemy przypisać do kolumny zbioru danych.

```
## tylko centrowanie: scale(DF$wiek, center = TRUE)
## tylko skalowanie:  scale(DF$wiek, center = FALSE)
## normalizacja
(DF$wiekNorm <- scale(DF$wiek))
```

```
##           [,1]
## [1,]  1.3555998
## [2,] -0.9071616
## [3,]  0.1213663
## [4,] -0.5698045
## attr(,"scaled:center")
## [1] 23.525
```



```
## attr(,"scaled:scale")
## [1] 12.15329
```

Otrzymujemy nowy, poszerzony zbiór danych

```
DF
```

```
##      imie  wiek numer      oczy  wiekNo
## 1  Maja  40.0     1    niebieskie  1.35559
## 2  Anna  12.5     2 jasno-niebieskie -0.90716
## 3  Zosia  25.0    NA      ciemne    0.12130
## 4  Anna  16.6     4    niebieskie -0.56980
```

## Usuwanie lub zastępowanie brakujących danych

W kolumnie `numer` występują brakujące wartości `NA`.

Takie wartości możemy usunąć (tzn. usunąć musimy cały wiersz) korzystając z funkcji `na.omit()`

```
na.omit(DF)
```

```
##      imie  wiek numer      oczy  wiekNo
## 1  Maja  40.0     1    niebieskie  1.35559
## 2  Anna  12.5     2 jasno-niebieskie -0.90716
## 4  Anna  16.6     4    niebieskie -0.56980
```

Lub zastąpić przez np. średnią w kolumnie. Na poniższym przykładzie funkcją `is.na()` identyfikuje wartości brakujące, funkcja `which()` wyznacza indeksy wartości

brakujących. Te indeksy będą następnie wykorzystane aby nadpisać wartości brakujące przez średnią wyznaczoną przez funkcję `mean()`.

```
(ktorePuste <- which(is.na(DF$numer)))  
  
## [1] 3  
  
DF$numer[ktorePuste] <- mean(DF$numer, na.rm=TRUE)
```

Otrzymujemy nowy, uzupełniony zbiór danych

DF					
##	imie	wiek	numer	oczy	wie
## 1	Maja	40.0	1.000000	niebieskie	1.39
## 2	Anna	12.5	2.000000	jasno-niebieskie	-0.90
## 3	Zosia	25.0	2.333333	ciemne	0.12
## 4	Anna	16.6	4.000000	niebieskie	-0.50

# Nazwy czynników w zmiennej jakościowej

Gdy przetwarzamy zmienną jakościową, częstą operacją jest zmiana nazw poziomów, zmiana ich kolejności lub połączenie kilku poziomów w jeden.

```
DF$oczy  
  
## [1] niebieskie jasno-niebieskie ciemne  
## Levels: ciemne jasno-niebieskie niebieskie
```

Aby wyświetlić lub zmienić nazwy poziomów można wykorzystać funkcję `levels()`. W podobny sposób co dla `colnames()` tą samą funkcją możemy wyświetlić wektor nazw lub go zmienić.

Jeżeli przypiszemy do niego wartości z powtarzającymi się napisami, to wskazane poziomy zostaną połączone w jeden.

```
levels(DF$oczy)
```

```
## [1] "ciemne" "jasno-niebieskie" "
```

```
levels(DF$oczy) <- c("ciemne", "niebieskie", "niebieskie")
DF$oczy
```

```
## [1] niebieskie niebieskie ciemne niebieskie
## Levels: ciemne niebieskie
```

Jeżeli chcemy zmienić kolejność poziomów w zmiennej jakościowej, to najłatwiej jest to zrobić z użyciem funkcji `factor()`. W argumencie `levels` możemy podać poziomy w dowolnej kolejności. Wartości są takie same, ale kolejność poziomów jest inna. Kolejność poziomów jest ważna w tworzeniu wykresów (odpowiada ona kolejności poziomów na wykresie) oraz w modelowaniu statystycznym.

```
factor(DF$oczy, levels=c("niebieskie", "ciemne", "ciemne"))
```

```
## [1] niebieskie niebieskie ciemne niebieskie
```

```
## Levels: niebieskie ciemne
```

Otrzymujemy nowy, zmieniony zbiór danych

```
DF
```

##		imie	wiek	numer	oczy	wiekNorm
##	1	Maja	40.0	1.000000	niebieskie	1.3555998
##	2	Anna	12.5	2.000000	niebieskie	-0.9071616
##	3	Zosia	25.0	2.333333	ciemne	0.1213663
##	4	Anna	16.6	4.000000	niebieskie	-0.5698045

## Podmiana dowolnej wartości

Często się zdarza, że w kolumnie podmienić trzeba wszystkie wartości na inne. Być może ktoś omyłkowo wpisał 100.00 zamiast 10000 i trzeba zastąpić jedną wartość przez drugą. Być może ktoś źle wpisywał nazwy terapii i chcemy te nazwy pozmienić.

Na przykładzie poniżej zastąpimy imiona Anna na Joanna. Wynik przyrównania zwraca wektor wartości logicznych.

```
DF$imie=="Anna"
```

```
## [1] FALSE TRUE FALSE TRUE
```

Można ten wektor wykorzystać do indeksowania wiersza. Jeżeli do kilkuelementowego wektora przypiszemy jedną wartość, tak jak w przykładzie poniżej, to wszystkie

elementy tego wektora zostaną zastąpione przez tę wartość.

```
DF$imie[DF$imie=="Anna"] <- "Joanna"
```

W wyniku tej instrukcji otrzymujemy nowy, zmieniony zbiór danych.

DF						
##		imie	wiek	numer	oczy	wiekNorm
## 1		Maja	40.0	1.000000	niebieskie	1.3555998
## 2		Joanna	12.5	2.000000	niebieskie	-0.9071616
## 3		Zosia	25.0	2.333333	ciemne	0.1213663
## 4		Joanna	16.6	4.000000	niebieskie	-0.5698045

## Zadania

Po wczytaniu poniższej ramki danych napraw następujące błędy:

- popraw nazwę kolumny `litary`,
- zamień wartości `X` i `Y` odpowiednio na `A` i `B`,
- zamień kolumnę `liczby` na kolumnę `liczb`,
- zastąp brakujące dane w kolumnie `braki`.

```
df <- data.frame(  
  litary = c("X", "B", "Y", "D"),  
  liczby = c("1", "2", "3", "0", "4", "0"),  
  braki = c(NA, 1, NA, 1),
```

```
stringsAsFactors = FALSE)
```

```
df
```

##	litary	liczby	braki
## 1	X	1	NA
## 2	B	2	1
## 3	Y	3, 0	NA
## 4	D	4, 0	1

## Co dalej

Na tych kilku slajdach zaledwie zarysowaliśmy temat czyszczenia danych. Więcej informacji na temat poprawiania zbiorów danych znaleźć można na przykład w pozycjach:

- Rozdział 3.3 „Wstępne przetwarzanie danych” w książce „Przewodnik po pakiecie R”, dostępnej pod adresem <http://biecek.pl/R/>
- W języku angielskim książka „An introduction to data cleaning with R” [http://cran.r-project.org/doc/contrib/de\\_Jonge+van\\_der\\_Loo-Introduction\\_to\\_data\\_cleaning\\_with\\_R.pdf](http://cran.r-project.org/doc/contrib/de_Jonge+van_der_Loo-Introduction_to_data_cleaning_with_R.pdf)
- W języku angielskim kurs na portalu Coursera „Getting and Cleaning Data” <https://www.coursera.org/course/getdata>

Przykładowe odpowiedzi znajdują się na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)



# Filtrowanie danych [dplyr / filter]

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 17*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Ptaki vs. koty](#)
- [Filtrowanie danych](#)
- [Dwa lub więcej warunków](#)
- [Auta](#)
- [Tabele danych](#)
- [Filtrowanie danych](#)
- [Dwa warunki](#)
- [Zadania](#)

## O czym jest ten odcinek

Bazy danych zawierają najczęściej znacznie więcej informacji niż jest nam potrzebne. Jedną z kluczowych operacji jest więc filtrowanie, wybieranie tylko tych



wierszy, które nas interesują. Przy czym stwierdzenie ‘interesują nas’ oznacza spełnianie pewnych kryteriów.

W procesie przygotowania danych takie filtrowanie jest niesamowicie częstą instrukcją. W odcinku 7 pokazaliśmy jak wybierać wiersze spełniające określone kryterium, używając operatora `[]`. W codziennej pracy nie tylko wygodniej, ale i szybciej jest wykorzystać dedykowaną funkcję `filter()` z pakietu `dplyr`.

W tym odcinku nauczymy się:

- jak ze zbioru danych wybierać tylko interesujące nas wiersze,
- jak tworzyć złożone kryteria wyboru wierszy.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki`, a drugi, znacznie większy, to `auta2012`. Oba dostępne są w pakiecie `PogromcyDanych`.

---

## Ptaki vs. koty

Rozpocznijmy przykład filtrowania od zbioru danych o kotach i ptakach. Ten zbiór danych jest dostarczany razem z pakietem `PogromcyDanych`. Jest on na tyle mały, że

wynik operacji będzie można przedstawić na ekranie.

Pokażemy jak z tego zbioru danych wybierać wiersze.  
Widząc cały zbiór danych łatwiej zrozumieć które wiersze zostały wyfiltrowane.

```
library(PogromcyDanych)
koty_ptaki
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	z
## 4		Puma	80.00	1.7	70	z
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	
## 10	Orzel przedni		5.00	0.9	160	
## 11	Sokol wedrowny		0.70	0.5	110	
## 12	Sokol norweski		2.00	0.7	100	
## 13	Albatros		4.00	0.8	120	Poln
##		wagaKategoria	habitat_napis			
## 1		(100,1e+03]	Azja			
## 2		(100,1e+03]	Afryka			
## 3		(10,100]	Ameryka			
## 4		(10,100]	Ameryka			
## 5		(10,100]	Azja			
## 6		(10,100]	Afryka			
## 7		(10,100]	Azja			
## 8		(0,1]	Euroazja			
## 9		(100,1e+03]	Afryka			
## 10		(1,10]	Polnoc			

##	11	(0, 1]	Polnoc
##	12	(1, 10]	Polnoc
##	13	(1, 10]	Poludnie

## Filtrowanie danych

Funkcja `filter()` redukuje liczbę wierszy, zostawiając tylko te, które spełniają określony warunek (jeden lub kilka). Pierwszym argumentem tej funkcji jest ramka danych (klasa `data.frame`) lub tabela danych (klasa `tbl`, przedstawimy ją na slajdzie 7). Kolejne argumenty (a może być ich jeden lub więcej) to warunki logiczne. Wynikiem jest ramka danych z wierszami, które spełniają jednocześnie wszystkie wskazane warunki.

Przykładowo, aby zobaczyć, które zwierzaki przekraczają prędkość 100 km na godzinę możemy użyć następujących komend. W pierwszej linii wczytujemy pakiet `dplyr`.

```
library(dplyr)
filter(koty_ptaki,
       predkosc > 100)
```

##		gatunek	waga	dlugosc	predkosc	hak
##	1	Gepard	60.00	1.4	115	A:
##	2	Jerzyk	0.05	0.2	170	Euro
##	3	Orzel przedni	5.00	0.9	160	Pe
##	4	Sokol wedrowny	0.70	0.5	110	Pe
##	5	Albatros	4.00	0.8	120	Poln

##	waga	Kategoria	habitat	napis
## 1	(10, 100]		Afryka	
## 2	(0, 1]		Euroazja	
## 3	(1, 10]		Polnoc	
## 4	(0, 1]		Polnoc	
## 5	(1, 10]		Poludnie	

## Dwa lub więcej warunków

Możemy określać jednocześnie więcej warunków dla filtrowania. Każdy kolejny warunek można podać jako kolejny argument funkcji `filter()`.

Poniższy przykład wybiera ze zbioru danych tylko te gatunki, które spełniają jednocześnie trzy warunki.

Są to ptaki, które przekraczają prędkość 100 km na godzinę i występują na półkuli północnej.

Operator `%in%` zwraca TRUE jeżeli określona wartość występuje w wektorze po prawej stronie.

```
filter(koty_ptaki,
       predkosc > 100,
       druzyna == "Ptak",
       habitat %in% c("Polnoc", "Euroazja"))
```

##	gatunek	waga	dlugosc	predkosc	hab:
## 1	Jerzyk	0.05	0.2	170	Euroaz

##	2	Orzel przedni	5.00	0.9	160	Po.
##	3	Sokol wedrowny	0.70	0.5	110	Po.
##		wagaKategoria	habitat	napis		
##	1		(0,1]	Euroazja		
##	2		(1,10]	Polnoc		
##	3		(0,1]	Polnoc		

## Auta

Operacje filtrowania przećwiczmy też na znacznie większym zbiorze danych, który już nie sposób w całości wyświetlić na ekranie, a mianowicie na zbiorze danych o cenach ofertowych używanych samochodów.

Zbiór danych `auta2012` jest dostępny po wczytaniu pakietu `PogromcyDanych`. Poniżej przedstawiamy 6 pierwszych wierszy z tego zbioru danych. Szczegółowy opis tego zbioru znajduje się na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

```
head(auta2012)
```

```
## Source: local data frame [6 x 28]
```

##		Cena	Waluta	Cena.w.PLN	Brutto.netto	KM
##	1	49900	PLN	49900	brutto	140
##	2	88000	PLN	88000	brutto	156
##	3	86000	PLN	86000	brutto	150
##	4	25900	PLN	25900	brutto	163
##	5	55900	PLN	55900	netto	NA

```
## 6 45900 PLN 45900 netto 150 :  
## Variables not shown: Wersja (fctr), Liczba.c  
## (dbl), Przebieg.w.km (dbl), Rodzaj.paliwa  
## Kolor (fctr), Kraj.aktualnej.rejestracji  
## (fctr), Pojazd.uszkodzony (fctr), Skrzynia  
## Status.pojazdu.sprowadzonego (fctr), Wypos  
## Rodzaj.paliwa.posortowany (fctr), Kolor_na  
## Wyposazenie.dodatkowe_napis (chr), czy_met  
## (lgl), szyby (lgl), MarkaModel (chr)
```

---

## Tabele danych

W tym miejscu warto zaznaczyć, że funkcje z pakietu `dplyr` pracują nie tylko na ramkach danych, które już znamy i które mają klasę `data.frame`, ale również na tabelach danych - obiektach klasy `tbl`.

Z użytkowego punktu widzenia, tabele danych od ramek danych różnią się głównie sposobem wyświetlania na ekranie (nie wyświetlają się kolumny nie mieszczące się w obecnej szerokości konsoli). W rzeczywistości różnią się także szeregiem innych właściwości, między innymi wewnętrzną reprezentacją, która pozwala na szybsze operowanie na danych w strukturze `tbl`.

Jeżeli mamy więc dane duże, o dużej liczbie wierszy lub kolumn, często dobrym pomysłem jest przekształcenie je na postać tabeli danych.

Ramkę danych na tabelę można zamienić funkcją `tbl_df` a tabelę na ramkę danych funkcją `as.data.frame()`.

```
auta2012 <- tbl_df(auta2012)
auta2012

## Source: local data frame [207,602 x 28]
##
##      Cena Waluta Cena.w.PLN Brutto.netto KM
##  1  49900    PLN      49900      brutto 140
##  2  88000    PLN      88000      brutto 156
##  3  86000    PLN      86000      brutto 150
##  4  25900    PLN      25900      brutto 163
##  5  55900    PLN      55900        netto NA
##  6  45900    PLN      45900        netto 150
##  7  39900    PLN      39900      brutto 115
##  8  38900    PLN      38900      brutto 115
##  9  24900    PLN      24900      brutto  68
## 10  79900    PLN      79900        netto 175
## .. ...
## Variables not shown: Wersja (fctr), Liczba.km (dbl), Przebieg.w.km (dbl), Rodzaj.paliwa (fctr), Kolor (fctr), Kraj.aktualnej.rejestracji (fctr), Pojazd.uszkodzony (fctr), Skrzynia (fctr), Status.pojazdu.sprawadzonego (fctr), Wyposazenie (fctr), Rodzaj.paliwa.posortowany (fctr), Kolor_nazwy (fctr), Wyposazenie.dodatkowe_napis (chr), czy_metalowy (lgl), szyby (lgl), MarkaModel (chr)
```

# Filtrowanie danych

Przypuśćmy, że chcemy ograniczyć listę aut do jednej

wybranej marki.

Przykładowo, jeżeli chcemy pozostawić tylko samochody marki Porsche, możemy użyć warunku

Marka == 'Porsche'.

```
tmp <- filter(auta2012,
               Marka == "Porsche")
head(tmp)
```

```
## Source: local data frame [6 x 28]
```

##		Cena	Waluta	Cena.w.PLN	Brutto.netto	KM
## 1	244900	PLN	244900	netto	388	
## 2	229000	PLN	229000	brutto	355	
## 3	133990	PLN	133990	netto	295	
## 4	154900	PLN	154900	brutto	295	
## 5	162520	PLN	162520	netto	500	
## 6	162520	PLN	162520	netto	500	

## Variables not shown: Wersja (fctr), Liczba. (dbl), Przebieg.w.km (dbl), Rodzaj.paliwa (fctr), Kraj.aktualnej.rejestracji (fctr), Pojazd.uszkodzony (fctr), Skrzynia (fctr), Status.pojazdu.sprowadzonego (fctr), Wypos. (dbl), Rodzaj.paliwa.posortowany (fctr), Kolor\_nazwy (fctr), Wyposazenie.dodatkowe\_napis (chr), czy\_metalowy (lgl), szyby (lgl), MarkaModel (chr)

## Dwa warunki

Możemy określać jednocześnie więcej warunków. Na



przykład tylko samochody Porsche o silnikach o liczbie koni mechanicznych przekraczających 300 możemy odfiltrować w następujący sposób.

```
tylkoPorscheZDuzymSilnikiem <- filter(auta2012,
  Marka == "Porsche",
  KM > 300)
head(tylkoPorscheZDuzymSilnikiem)
```

```
## Source: local data frame [6 x 28]
##
##      Cena Waluta Cena.w.PLN Brutto.netto KM
## 1 244900    PLN    244900      netto 388
## 2 229000    PLN    229000      brutto 355
## 3 162520    PLN    162520      netto 500
## 4 162520    PLN    162520      netto 500
## 5  69900    PLN     69900      brutto 340
## 6 359000    PLN   359000      brutto 400
## Variables not shown: Wersja (fctr), Liczba.cylindrow (dbl),
## Przebieg.w.km (dbl), Rodzaj.paliwa (fctr), Kolor (fctr),
## Kraj.aktualnej.rejestracji (fctr), Pojazd.uszkodzony (fctr),
## Skrzynia (fctr), Status.pojazdu.sprowadzonego (fctr), Wyposazenie (dbl),
## Rodzaj.paliwa.posortowany (fctr), Kolor_niebieski (fctr),
## Wyposazenie.dodatkowe_napis (chr), czy_mechaniczny (fctr),
## Wersja (fctr), szyby (lgl), MarkaModel (chr)
```

---

## Zadania

- Ze zbioru danych `auta2012` pozostaw tylko samochody o wieku do pięciu lat (czyli o roku

produkcji > 2007). Nazwij ten wynikowy zbiór danych `młodeAuta`

- Z przed chwilą stworzonego zbioru danych `młodeAuta` wybierz tylko Fiaty 500. Ile jest takich aut? (wymiary ramki danych możesz sprawdzić funkcją `dim()`)

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Sortowanie danych [dplyr / arrange]

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 18*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Ptaki vs. koty](#)
- [Sortowanie danych](#)
- [Sortowanie danych](#)
- [Sortowanie po dwóch zmiennych](#)
- [Auta](#)
- [Najtańsze i najdroższe Porsche](#)
- [Model i cena](#)
- [Sortowanie malejąco](#)
- [Zadania](#)

## O czym jest ten odcinek

W programie R wiersze można sortować na kilka sposobów, jeden z użyciem funkcji `order()` pokazaliśmy

w odcinku 7.

W codziennej pracy z danymi nie tylko wygodniej, ale i szybciej jest wykorzystać dedykowaną funkcję `arrange()` z pakietu `dplyr`.

W tym odcinku nauczymy się:

- jak porządkować wiersze rosnąco / malejąco względem określonej kolumny,
- jak i po co sortować po więcej niż jednej kolumnie.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012`, oba dostępne w pakiecie `PogromcyDanych`.

---

## Ptaki vs. koty

Przykłady dla sortowania rozpoczniemy na niewielkim zbiorze danych o kotach i ptakach. Ten zbiór danych jest dostarczany razem z pakietem `PogromcyDanych`. Jest on na tyle mały, że łatwo ogarnąć go wzrokiem i przekonać się w jaki sposób zadziałało sortowanie wierszy.

```
library(PogromcyDanych)
koty_ptaki
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	2
## 4		Puma	80.00	1.7	70	2
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	
## 10	Orzel	przedni	5.00	0.9	160	
## 11	Sokol	wedrowny	0.70	0.5	110	
## 12	Sokol	norweski	2.00	0.7	100	
## 13		Albatros	4.00	0.8	120	Pe
##		wagaKategoria	habitat_napis			
## 1		(100,1e+03]	Azja			
## 2		(100,1e+03]	Afryka			
## 3		(10,100]	Ameryka			
## 4		(10,100]	Ameryka			
## 5		(10,100]	Azja			
## 6		(10,100]	Afryka			
## 7		(10,100]	Azja			
## 8		(0,1]	Euroazja			
## 9		(100,1e+03]	Afryka			
## 10		(1,10]	Polnoc			
## 11		(0,1]	Polnoc			
## 12		(1,10]	Polnoc			
## 13		(1,10]	Poludnie			

# Sortowanie danych

O ile wektor wartości można posortować funkcją `sort()`,

to do sortowania wierszy w ramce danych względem określonej kolumny wygodnie jest użyć funkcji `arrange()`.

Ta funkcja jako pierwszy argument przyjmuje ramkę danych, a jako kolejne argumenty przyjmuje zmienne, wzdłuż których dane są sortowane.

Przykładowo aby posortować wiersze w zbiorze `koty_ptaki` w kolejności rosnącej ze względu na prędkość (kolumna `predkosc`) można użyć następującej komendy.

```
library(dplyr)
arrange(koty_ptaki, predkosc)
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Irbis	50.00	1.3	65	
## 3		Puma	80.00	1.7	70	z
## 4		Strus	150.00	2.5	70	
## 5		Lew	200.00	2.0	80	
## 6		Leopard	70.00	1.4	85	
## 7		Jaguar	100.00	1.7	90	z
## 8	Sokol	norweski	2.00	0.7	100	
## 9	Sokol	wedrowny	0.70	0.5	110	
## 10		Gepard	60.00	1.4	115	
## 11		Albatros	4.00	0.8	120	Pe
## 12	Orzel	przedni	5.00	0.9	160	
## 13		Jerzyk	0.05	0.2	170	Et
##		wagaKategoria	habitat_napis			
## 1		(100,1e+03]		Azja		

## 2	(10,100]	Azja
## 3	(10,100]	Ameryka
## 4	(100,1e+03]	Afryka
## 5	(100,1e+03]	Afryka
## 6	(10,100]	Azja
## 7	(10,100]	Ameryka
## 8	(1,10]	Polnoc
## 9	(0,1]	Polnoc
## 10	(10,100]	Afryka
## 11	(1,10]	Poludnie
## 12	(1,10]	Polnoc
## 13	(0,1]	Euroazja

## Sortowanie danych

Aby posortować w kolejności malejącej można albo zmienną, po której ma nastąpić sortowanie opatrzyć funkcją `desc()`, albo - dla zmiennych liczbowych - dodać znak `-` przed zmienną.

Obie poniższe instrukcje mają taki sam efekt.

```
arrange(koty_ptaki, desc(predkosc))
```

##	gatunek	waga	dlugosc	predkosc	1
## 1	Jerzyk	0.05	0.2	170	En
## 2	Orzel przedni	5.00	0.9	160	
## 3	Albatros	4.00	0.8	120	Pe
## 4	Gepard	60.00	1.4	115	
## 5	Sokol wedrowny	0.70	0.5	110	
## 6	Sokol norweski	2.00	0.7	100	

##	7	Jaguar	100.00	1.7	90	z
##	8	Leopard	70.00	1.4	85	
##	9	Lew	200.00	2.0	80	
##	10	Puma	80.00	1.7	70	z
##	11	Strus	150.00	2.5	70	
##	12	Irbis	50.00	1.3	65	
##	13	Tygrys	300.00	2.5	60	

##	wagaKategoria		habitat_napis	
##	1	(0,1]	Euroazja	
##	2	(1,10]	Polnoc	
##	3	(1,10]	Poludnie	
##	4	(10,100]	Afryka	
##	5	(0,1]	Polnoc	
##	6	(1,10]	Polnoc	
##	7	(10,100]	Ameryka	
##	8	(10,100]	Azja	
##	9	(100,1e+03]	Afryka	
##	10	(10,100]	Ameryka	
##	11	(100,1e+03]	Afryka	
##	12	(10,100]	Azja	
##	13	(100,1e+03]	Azja	

arrange(koty\_ptaki, -predkosc)

##	gatunek		waga	dlugosc	predkosc	l
##	1	Jerzyk	0.05	0.2	170	Eu
##	2	Orzel przedni	5.00	0.9	160	
##	3	Albatros	4.00	0.8	120	Pe
##	4	Gepard	60.00	1.4	115	
##	5	Sokol wedrowny	0.70	0.5	110	
##	6	Sokol norweski	2.00	0.7	100	
##	7	Jaguar	100.00	1.7	90	z
##	8	Leopard	70.00	1.4	85	
##	9	Lew	200.00	2.0	80	
##	10	Puma	80.00	1.7	70	z



##	11	Strus	150.00	2.5	70
##	12	Irbis	50.00	1.3	65
##	13	Tygrys	300.00	2.5	60
##	wagaKategoria habitat_napis				
##	1	(0,1]	Euroazja		
##	2	(1,10]	Polnoc		
##	3	(1,10]	Poludnie		
##	4	(10,100]	Afryka		
##	5	(0,1]	Polnoc		
##	6	(1,10]	Polnoc		
##	7	(10,100]	Ameryka		
##	8	(10,100]	Azja		
##	9	(100,1e+03]	Afryka		
##	10	(10,100]	Ameryka		
##	11	(100,1e+03]	Afryka		
##	12	(10,100]	Azja		
##	13	(100,1e+03]	Azja		

## Sortowanie po dwóch zmiennych

Jeżeli dla zmiennej, wzdłuż której sortujemy, mogą wystąpić remisy, czyli te same wartości, wtedy kolejność wierszy jest nieokreślona. Wygodnie jest w takich sytuacjach wskazać drugorzędowe kryterium sortowania.

Przykładowo, aby posortować wiersze uwzględniając prędkość, ale zrobić to osobno dla kotów osobno dla ptaków, możemy wskazać drużynę jako główne kryterium sortowania, a prędkość jako kryterium drugorzędne.

Sortując po kolumnie `druzyna` wiele wierszy ma te same wartości i o ich końcowej kolejności decyduje zmienna `predkosc`.

```
arrange(koty_ptaki,
        druzyzna, predkosc)
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Irbis	50.00	1.3	65	
## 3		Puma	80.00	1.7	70	z
## 4		Lew	200.00	2.0	80	
## 5		Leopard	70.00	1.4	85	
## 6		Jaguar	100.00	1.7	90	z
## 7		Gepard	60.00	1.4	115	
## 8		Strus	150.00	2.5	70	
## 9	Sokol	norweski	2.00	0.7	100	
## 10	Sokol	wedrowny	0.70	0.5	110	
## 11		Albatros	4.00	0.8	120	Pe
## 12	Orzel	przedni	5.00	0.9	160	
## 13		Jerzyk	0.05	0.2	170	En
##		waga	Kategoria	habitat	napis	
## 1		(100,1e+03]		Azja		
## 2		(10,100]		Azja		
## 3		(10,100]		Ameryka		
## 4		(100,1e+03]		Afryka		
## 5		(10,100]		Azja		
## 6		(10,100]		Ameryka		
## 7		(10,100]		Afryka		
## 8		(100,1e+03]		Afryka		
## 9		(1,10]		Polnoc		
## 10		(0,1]		Polnoc		
## 11		(1,10]		Poludnie		
## 12		(1,10]		Polnoc		

# Auta

Operacje sortowania przećwiczmy też na znacznie większym zbiorze danych, który już nie sposób w całości wyświetlić na ekranie. Mianowicie na zbiorze danych o cenach ofertowych używanych samochodów.

Zbiór danych `auta2012` jest dostępny po wczytaniu pakietu `PogromcyDanych`. Poniżej przedstawiamy 6 pierwszych wierszy z tego zbioru danych. Szczegółowy opis tego zbioru znajduje się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

Wyberzmy ze zbioru danych tylko samochody marki `Porsche` z silnikiem ponad 300 KM, posortujmy wiersze w tym zbiorze danych zaczynając od najtańszych.

```
tylkoPorscheZDuzymSilnikiem <- filter(auta2012,  
  Marka == "Porsche",  
  KM > 300)  
  
posortowanePorsche <-  
  arrange(tylkoPorscheZDuzymSilnikiem,  
    Cena.w.PLN)
```

---

# Najtańsze i najdroższe Porsche

Aby zobaczyć jakie wiersze są pierwsze a jakie są ostatnie w tym zbiorze danych, możemy użyć funkcji `head()` i `tail()`. Domyślnie przedstawianych jest pierwszych lub ostatnich 6 wierszy.

```
head(posortowanePorsche)
```

```
## Source: local data frame [6 x 28]
```

# #

##	Cena	Waluta	Cena.w.PLN	Brutto.netto	KM
## 1	1184	PLN	1184	brutto	350
## 2	1325	PLN	1325	brutto	340
## 3	2000	PLN	2000	brutto	500
## 4	4500	PLN	4500	brutto	450
## 5	9100	PLN	9100	brutto	521
## 6	10999	PLN	10999	netto	500
##	Variables not shown: Wersja (fctr), Liczba.km (dbl), Przebieg.w.km (dbl), Rodzaj.paliwa (fctr), Kolor (fctr), Kraj.aktualnej.rejestracji (fctr), Pojazd.uszkodzony (fctr), Skrzynia (fctr), Status.pojazdu.sprowadzonego (fctr), Wyposazenie (fctr), Rodzaj.paliwa.posortowany (fctr), Kolor.naprawy (fctr), Wyposazenie.dodatkowe_napis (chr), czy_metal (lgl), szyby (lgl), MarkaModel (chr)				

```
tail (posortowanePorsche)
```

```
## Source: local data frame [6 x 28]
```

##

##	Cena	Waluta	Cena.w.PLN	Brutto.netto	KM
## 1	999999	PLN	999999	brutto	500

##	2	249797	EUR	1103303	brutto	550
##	3	250335	EUR	1105680	brutto	550
##	4	325000	USD	1110655	brutto	612
##	5	1123466	PLN	1123466	brutto	550
##	6	1140674	PLN	1140674	brutto	550
##	Variables not shown: Wersja (fctr), Liczba.o					
##	(dbl), Przebieg.w.km (dbl), Rodzaj.paliwa					
##	Kolor (fctr), Kraj.aktualnej.rejestracji					
##	(fctr), Pojazd.uszkodzony (fctr), Skrzynia					
##	Status.pojazdu.sprawadzonego (fctr), Wypos					
##	Rodzaj.paliwa.posortowany (fctr), Kolor_na					
##	Wyposazenie.dodatkowe_napis (chr), czy_met					
##	(lgl), szyby (lgl), MarkaModel (chr)					

---

## Model i cena

Jeżeli wskażemy więcej niż jedną zmienną do sortowania, to w pierwszym kroku sortowanie odbędzie się wzdłuż pierwszej zmiennej, a w przypadku remisu (takich samych wartości), w drugim kroku, wzdłuż drugiej zmiennej.

W poniższym przypadku w pierwszym kroku sortujemy wzdłuż zmiennej `Model` auta, a gdy samochody mają ten sam model to ich kolejność jest określona przez drugą zmienną, czyli w tym przypadku cenę.

Zmienna `Model` za wartości przyjmuje napisy, a w tym przypadku sortowanie wykonywane jest w kolejności alfabetycznej.

```
head(
  arrange(tylkoPorscheZDuzymSilnikiem,
    Model, Cena.w.PLN)
)
```

```
## Source: local data frame [6 x 28]
##
##   Cena Waluta Cena.w.PLN Brutto.netto KM
## 1  2000     PLN   2000.00      brutto 500
## 2 13501     EUR  59631.22      brutto 305
## 3 60000     PLN  60000.00      brutto 325
## 4 66900     PLN  66900.00      brutto 320
## 5 71900     PLN  71900.00      brutto 320
## 6 16500     EUR  72877.20      brutto 530
## Variables not shown: Liczba.drzwi (fctr), Po
## Przebieg.w.km (dbl), Rodzaj.paliwa (fctr),
## (fctr), Kraj.aktualnej.rejestracji (fctr),
## Pojazd.uszkodzony (fctr), Skrzynia.biegow
## Status.pojazdu.sprowadzonego (fctr), Wypos
## Rodzaj.paliwa.posortowany (fctr), Kolor_na
## Wyposazenie.dodatkowe_napis (chr), czy_mel
## (lgl), szyby (lgl), MarkaModel (chr)
```

## Sortowanie malejąco

Domyślnie wartości sortowane są rosnąco. Jeżeli chcemy zmienić kolejność sortowania i sortować malejąco, to należy daną zmienną opakować funkcją `desc()`, tak jak na poniższym przykładzie (sortować można również po większej liczbie kolumn, np. walucie, marce i cenie).

```
head(
  arrange(tylkoPorscheZDuzymSilnikiem,
    Model, desc(Cena.w.PLN))
)

## Source: local data frame [6 x 28]
##
##      Cena Waluta Cena.w.PLN Brutto.netto KM
## 1 4488380    CZK   767961.8      netto 700
## 2 4221810    CZK   722351.7      netto 700
## 3 660000     PLN   660000.0      brutto 530
## 4 606432     PLN   606432.0      brutto 385
## 5 604000     PLN   604000.0      brutto 385
## 6 599000     PLN   599000.0      netto 530
## Variables not shown: Liczba.drzwi (fctr), Przebieg.w.km (dbl), Rodzaj.paliwa (fctr),
##      (fctr), Kraj.aktualnej.rejestracji (fctr), Pojazd.uszkodzony (fctr), Skrzynia.biegow
##      Status.pojazdu.sprowadzonego (fctr), Wypos
##      Rodzaj.paliwa.posortowany (fctr), Kolor_na
##      Wyposazenie.dodatkowe_napis (chr), czy_me
##      (lgl), szyby (lgl), MarkaModel (chr)
```

## Zadania

- Posortuj auta wzdłuż mocy silnika (liczby koni mechanicznych, kolumna `KM`) lub pojemności skokowej (kolumna `Pojemnosc.skokowa`). Które marki mają największe silniki?
- Wybierz tylko auta marki ‘Rolls-Royce’ i posortuj je po cenie.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)



# Potokowe przetwarzanie danych

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 19*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Przetwarzanie „krok po kroku”](#)
- [Przetwarzanie „na wielką cebulkę”](#)
- [Przetwarzanie potokowe](#)
- [Przetwarzanie potokowe - jak to czytać](#)
- [Wszystko płynie](#)
- [Co można zrobić z potokiem](#)
- [Zadania](#)

## O czym jest ten odcinek

Przetwarzanie danych składa się najczęściej z wielu kroków. Dane się filtruje, sortuje, grupuje, podsumowuje, znów filtruje, łączy się z innymi itd.

Taka sekwencja operacji przypomina potok, w którym dane wypływają ze źródła, a następnie trafiają na kolejne etapy przetwarzania.

Relatywnie niedawno do programu R wprowadzono operator pozwalający na łatwe opisywanie takich potoków przetwarzania. Można bez niego żyć, ale gdy już się go pozna i zacznie z nim pracować, życie bez niego straci na smaku.

W tym odcinku nauczymy się:

- jak składać ze sobą wywołania funkcji do przetwarzania danych w potoki przetwarzania,
- jakim instrukcjom operator przetwarzania potokowego jest równoważny.

Do ilustracji tych zagadnień wykorzystamy zbiór danych `auta2012` dostępny w pakiecie `PogromcyDanych`.

---

## Przetwarzanie „krok po kroku”

Jak już wspomnieliśmy analiza danych przez większość czasu polega na przetwarzaniu danych w tą i z powrotem.

Przyjrzyjmy się przykładowej serii operacji. Zaczniemy od danych o wszystkich ofertach sprzedaży samochodów,

odfiltrujemy samochody, które nie są Volkswagenami, pozostałe samochody posortujemy po cenie. Następnie pozostawimy tylko Golfy IV o przebiegu poniżej 50000 km.

Poniższa instrukcja nie jest najkrótszą z możliwych, celowo została rozciągnięta by łatwiej było przedstawić zalety potoków.

```
library(PogromcyDanych)
library(dplyr)
## tylko volkswagen
tylkoVolkswagen <- filter(auta2012,
                           Marka == "Volkswagen")
## posortowane
posortowaneVolkswagen <- arrange(tylkoVolkswagen,
                                   Cena.w.PLN)
## tylko Golf VI
tylkoGolfIV <- filter(posortowaneVolkswagen,
                      Model == "Golf", Wersja == "VI")
## tylko z małym przebiegiem
tylkoMalyPrzebieg <- filter(tylkoGolfIV,
                             Przebieg.w.km < 50000)
```

Ta sekwencja operacji tworzy cztery pomocnicze zbiory danych: `tylkoVolkswagen`, `posortowaneVolkswagen`, `tylkoGolfIV`, `tylkoMalyPrzebieg`.

Jeżeli potrzebujemy tylko ostatniego, to pozostałe wyłącznie zaśmiecają pamięć, najlepiej byłoby je od razu usunąć lub w ogóle nie tworzyć.

# Przetwarzanie „na wielką cebulkę”

Ponieważ w R wyniki jednej funkcji można przekazać do innej funkcji, to również wymienione operacje można złożyć w jedną wielką cebulkę.

W jedno olbrzymie wywołanie funkcji w funkcji przekazujące wyniki jednej funkcji bezpośrednio do kolejnej.

Taka wielka cebulka wyglądałaby następująco.

```
tylkoMalyPrzebieg <-  
  filter(  
    filter(  
      arrange(  
        filter(  
          auta2012,  
          Marka == "Volkswagen"),  
          Cena.w.PLN),  
        Model == "Golf", Wersja == "IV"),  
        Przebieg.w.km < 50000)
```

W tym rozwiązaniu nie są tworzone zbędne zmienne, ale sam zapis jest bardzo nieczytelny.

Nawet stosując wcięcia trudno nam zauważyć, które argumenty są do której funkcji.

Gdy przetwarzanie jest bardziej złożone to i taki blok może się jeszcze bardziej rozrastać, a tym samym trudniej będzie zrozumieć co się w kodzie dzieje.

---

## Przetwarzanie potokowe

Rozwiązaniem tego problemu jest stosowanie specjalnego operatora do przetwarzania potokowego `%>%`. Ten operator pochodzi z pakietu `magrittr` (cytując z jego dokumentacji: *to be pronounced with a sophisticated french accent*) i jest dostępny po włączeniu pakietu `dplyr` (jako pakiet zależny).

Jak działa ten operator?

Przekazuje lewą stronę operatora jako pierwszy argument prawej strony tego operatora.

Tak więc instrukcja `a %>% f(b)` jest równoważna instrukcji `f(a, b)`.

Ta prosta sztuczka pozwala znacząco skrócić zapis i uczynić go znacznie czytelniejszym.

```
tylkoMalyPrzebieg <-  
  auta2012 %>%  
  filter(Marka == "Volkswagen") %>%
```

```
arrange(Cena.w.PLN) %>%  
filter(Model == "Golf", Wersja == "IV") %>%  
filter(Przebieg.w.km < 50000)
```

```
head(tylkoMalyPrzebieg)
```

```
## Source: local data frame [6 x 28]
```

```
##
```

##		Cena	Waluta	Cena.w.PLN	Brutto.netto	KM	
## 1	4800	PLN	4800	brutto	150	1	
## 2	7500	PLN	7500	brutto	75	9	
## 3	8000	PLN	8000	brutto	100	7	
## 4	8300	PLN	8300	brutto	NA	1	
## 5	8500	PLN	8500	brutto	75	9	
## 6	8500	PLN	8500	brutto	100	7	

```
## Variables not shown: Liczba.drzwi (fctr), Po  
## Przebieg.w.km (dbl), Rodzaj.paliwa (fctr),  
## (fctr), Kraj.aktualnej.rejestracji (fctr),  
## Pojazd.uszkodzony (fctr), Skrzynia.biegow  
## Status.pojazdu.sprawadzonego (fctr), Wypos  
## Rodzaj.paliwa.posortowany (fctr), Kolor_na  
## Wyposazenie.dodatkowe_napis (chr), czy_met  
## (lgl), szyby (lgl), MarkaModel (chr)
```

---

## Przetwarzanie potokowe - jak to czytać

Jak czytać taki fragment?

Jest to znacznie prostsze ponieważ argumenty są blisko nazw funkcji, możemy więc czytać opis przetwarzania zdanie po zdaniu.

Przedstawiony przykład można czytać następująco.

Weź zbiór danych `auta2012`,

następnie zastosuj funkcję `filter` pozostawiając tylko  
auta `Marka == "Volkswagen"`,

następnie posortuj auta malejąco wzdułuż zmiennej  
`Cena.w.PLN`,

następnie zastosuj funkcję `filter` pozostawiając tylko  
auta o modelu Golf w wersji IV,

następnie zastosuj funkcję `filter` pozostawiając tylko  
auta o przebiegu poniżej 50 tys. km.

Zapis z operatorem `%>%` jest w wielu sytuacjach znacznie  
czytelniejszy i będziemy z niego często korzystać przy  
przetwarzaniu danych.

```
tylkoMalyPrzebieg <-  
  auta2012 %>%  
  filter(Marka == "Volkswagen") %>%  
  arrange(Cena.w.PLN) %>%  
  filter(Model == "Golf", Wersja == "IV") %>%  
  filter(Przebieg.w.km < 50000)
```

## Wszystko płynie

Jeżeli chcielibyśmy otrzymać pełny przepływ w czytaniu od góry na dół potoku, to zamiast operatora przypisania `<-` możemy wykorzystać operator `->` o identycznym znaczeniu a różniący się tylko kolejnością argumentów.

W poniższym kodzie zostanie wykonane przetwarzanie a w ostatnim kroku wynik zostanie przypisany do zmiennej `tylkoMalyPrzebieg`.

```
auta2012 %>%  
  filter(Marka == "Volkswagen") %>%  
  arrange(Cena.w.PLN) %>%  
  filter(Model == "Golf", Wersja == "IV") %>%  
  filter(Przebieg.w.km < 50000) ->  
tylkoMalyPrzebieg
```

## Co można zrobić z potokiem

Operator `%>%` jest najczęściej wykorzystywany w pracy z funkcjami z pakietów `dplyr` i `ggvis`, ale można go oczywiście stosować również do innych funkcji. Koniec końców, jedyne co robi, to przekazuje swój lewy argument jako pierwszy argument wyrażenia po prawej stronie.

Przykładowo, aby wykorzystać funkcję `dim()` do sprawdzenia wymiarów ramki `tylkoMalyPrzebieg`.

```
tylkoMalyPrzebieg %>% dim()
```



Aby wyświetlić kilka pierwszych wierszy z ramki danych `tylkoMalyPrzebieg`.

```
tylkoMalyPrzebieg %>% head()
```

Jeżeli chcielibyśmy przekazać zbiór danych z lewej strony operatora `%>%` w inne miejsce niż pierwszy argument, również możemy to zrobić. Miejsce w które ma być wstawiony argument należy w tym celu zamarkować znakiem kropki `.` (kropki w nazwach zmiennych się nie liczą).

W poniższym przykładzie używamy funkcji `lm()`, podajemy jako pierwszy argument formułę, a jako drugi argument zbiór danych. W tym przykładzie, ponieważ zbiór danych to `.`, to będzie za niego wstawiony zbiór `tylkoMalyPrzebieg` (lewa strona operatora `%>%`).

```
tylkoMalyPrzebieg %>% lm(Cena.w.PLN ~ Przebieg.w.km)

##
## Call:
## lm(formula = Cena.w.PLN ~ Przebieg.w.km, data = tylkoMalyPrzebieg)
##
## Coefficients:
##      (Intercept)      Przebieg.w.km
##      1.579e+04          6.221e-02
```

---

## Zadania

- Użyj operatora `%>%` by ze zbioru danych `auta2012` wybrać tylko 10 najtańszych aut marki Rolls-Royce.
- Użyj operatora `%>%` by ze zbioru danych `auta2012` wybrać 5 Volkswagenów o największych silnikach.

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Wybór zmiennych z danych [dplyr / select]

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 20*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Wybór zmiennych przez wskazanie](#)
- [Wybór zmiennych używając operatora : lub -](#)
- [Auta](#)
- [Wybór zmiennych przez wskazanie](#)
- [Wybór zmiennych przez usunięcie](#)
- [Wybór zmiennych przez wzorzec](#)
- [Zadania](#)

## O czym jest ten odcinek

Pracując na danych o ofertach sprzedaży aut można było odczuć jak niewygodna jest praca na danych które mają wiele kolumn, w sytuacji gdy potrzebujemy jedynie kilku z nich. W odcinku 7 pokazaliśmy jak wybierać kolumny

spełniające określone kryterium, używając operatora `[]`.

W codziennej pracy aby wybrać kolumny z całego zbioru danych nie tylko wygodniej, ale i szybciej jest wykorzystać dedykowaną funkcję `select()` z pakietu `dplyr`.

W tym odcinku nauczymy się:

- jak ze zbioru danych wybierać tylko interesujące nas kolumny,
- jak ze zbioru danych wybierać kolumny wszystkie poza wskazanymi.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012`, oba dostępne w pakiecie `PogromcyDanych`.

---

## Wybór zmiennych przez wskazanie

Zbiór `koty_ptaki` na niewiele kolumn, można je wszystkie wyświetlić na konsoli. Czasem jednak nadmiarowe zmienne jedynie utrudniają skupienie się na tym co ważne.

Aby zmienić kolejność kolumn lub wybrać jedynie

podzbiór interesujących kolumn wykorzystać możemy funkcję `select()`.

W przykładzie poniżej wyświetlimy nazwy kolumn w zbiorze danych `koty_ptaki`. A następnie wybierzemy trzy z tych kolumn w kolejności `gatunek`, `predkosc`, `waga`.

```
library(PogromcyDanych)
library(dplyr)
colnames(koty_ptaki)

## [1] "gatunek"          "waga"              "dlugosc"
## [5] "habitat"          "zywotnosc"         "druzyna"
## [9] "habitat_napis"
```

```
koty_ptaki %>%
  select(gatunek, predkosc, waga)
```

##		gatunek	predkosc	waga
## 1		Tygrys	60	300.00
## 2		Lew	80	200.00
## 3		Jaguar	90	100.00
## 4		Puma	70	80.00
## 5		Leopard	85	70.00
## 6		Gepard	115	60.00
## 7		Irbis	65	50.00
## 8		Jerzyk	170	0.05
## 9		Strus	70	150.00
## 10	Orzel	przedni	160	5.00
## 11	Sokol	wedrowny	110	0.70
## 12	Sokol	norweski	100	2.00
## 13		Albatros	120	4.00



# Wybór zmiennych używając operatora : lub -

Jeżeli kolumn w zbiorze danych jest wiele to wymienienie ich nazw jedna po drugiej może być niewygodne. W tym przypadku warto wykorzystać operator `:`, który pozwala na wybór kolumn od wskazanej do wskazanej.

```
koty_ptaki %>%  
  select(gatunek:dlugosc, druzyna) %>%  
  head()
```

##	gatunek	waga	dlugosc	druzyna
## 1	Tygrys	300	2.5	Kot
## 2	Lew	200	2.0	Kot
## 3	Jaguar	100	1.7	Kot
## 4	Puma	80	1.7	Kot
## 5	Leopard	70	1.4	Kot
## 6	Gepard	60	1.4	Kot

W funkcji `select()` można wykorzystać operator `-` aby wybrać z ramki danych wszystkie kolumny poza wskazanymi.

```
koty_ptaki %>%  
  select(-habitat, -waga, -druzyna) %>%  
  head()
```

##	gatunek	dlugosc	predkosc	zywotnosc	wagaKat
## 1	Tygrys	2.5	60	25	(100,
## 2	Lew	2.0	80	29	(100,

##	3	Jaguar	1.7	90	15	(
##	4	Puma	1.7	70	13	(
##	5	Leopard	1.4	85	21	(
##	6	Gepard	1.4	115	12	(

# Auta

Operacje filtrowania przećwiczmy też na znacznie większym zbiorze danych, który już nie sposób w całości wyświetlić na ekranie. Mianowicie na zbiorze danych o cenach ofertowych używanych samochodów.

Zbiór danych `auta2012` jest dostępny po wczytaniu pakietu `PogromcyDanych`. Poniżej przedstawiamy listę nazw kolumn z tego zbioru danych, więcej informacji o tym zbiorze danych znaleźć można na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

W tym zbiorze dostępne są następujące dane.

```
colnames(auta2012)
```

##	[1]	"Cena"	"Waluta"
##	[3]	"Cena.w.PLN"	"Brutto"
##	[5]	"KM"	"kW"
##	[7]	"Marka"	"Model"
##	[9]	"Wersja"	"Liczba"
##	[11]	"Pojemnosc.skokowa"	"Przebieg"
##	[13]	"Rodzaj.paliwa"	"Rok.produkcji"
##	[15]	"Kolor"	"Kraj.albo_miejscowosc_produkcji"

```
## [17] "Kraj.pochodzenia" "Pojazd
## [19] "Skrzynia.biegow" "Status
## [21] "Wyposazenie.dodatkowe" "Rodzaj
## [23] "Kolor_napis" "Wyposa
## [25] "czy_metallic" "maKlima
## [27] "szyby" "MarkaMo
```

Funkcja `select()` pozwala w prosty sposób wybrać kilka potrzebnych zmiennych z dużego zbioru danych.

Pierwszym argumentem jest ramka danych, a kolejne to zmienne, które chcemy pozostawić.

## Wybór zmiennych przez wskazanie

Przypuśćmy, że chcemy wybrać tylko Cenę, Markę, Model, Wersję i Przebieg.w.km. Wystarczy te zmienne wymienić jako argumenty funkcji `select()`.

```
auta2012 %>%
  select(Cena.w.PLN, Marka, Model, Wersja, Przebieg.w.km) %>%
  head()
```

```
## Source: local data frame [6 x 5]
```

```
##
##   Cena.w.PLN      Marka      Model Wersja
## 1      49900        Kia    Carens
## 2      88000  Mitsubishi Outlander
## 3      86000   Chevrolet  Captiva
## 4      25900        Volvo      S80
## 5      55900 Mercedes-Benz  Sprinter
```



# Wybór zmiennych przez usunięcie

Jeżeli chcemy usunąć kilka kolumn, można do tego wykorzystać operator `-`.

Na poniższym przykładzie usuwamy 9 wskazanych kolumn.

```
auta2012 %>%
  select(-Cena, -Waluta, -Brutto.netto, -Rodzaj,
         -Kraj.aktualnej.rejestracji, -Kraj.poc,
         -Pojazd.uszkodzony, -Skrzynia.biegow,
         -Status.pojazdu.sprawadzonego, -Wyposazenie)
head()
```

##	Source: local data frame [6 x 18]					
##						
##		Cena.w.PLN	KM	kW	Marka	Model
## 1		49900	140	103	Kia	Caren
## 2		88000	156	115	Mitsubishi	Outlander
## 3		86000	150	110	Chevrolet	Captiva
## 4		25900	163	120	Volvo	S80
## 5		55900	NA	NA	Mercedes-Benz	Sprinter
## 6		45900	150	110	Mercedes-Benz	Viano
##	Variables not shown: Pojemnosc.skokowa (dbl),					
##	Rok.produkcji (dbl), Kolor (fctr), Rodzaj					
##	Kolor_napis (chr), Wyposazenie.dodatkowe (lgl),					
##	(lgl), maKlimatyzacje (lgl), szyby (lgl),					

# Wybór zmiennych przez wzorzec

Wybierając kolumny, można również wskazywać te, których nazwy pasują do zadanego wzorca.

Przykład poniżej wykorzystujemy funkcję `matches()` aby wybrać wszystkie nazwy kolumn, które w nazwie mają wzorzec `aj`.

```
auta2012 %>%
  select(matches("aj")) %>%
  head()

## Source: local data frame [6 x 4]
##
##   Rodzaj.paliwa Kraj.aktualnej.rejestracji i
## 1             H                      Polska
## 2             H                      Polska
## 3             H                      Polska
## 4             H                      Polska
## 5             H
## 6             H
## Variables not shown: Rodzaj.paliwa.posortowa
```

---

## Zadania

- Wybierz tylko samochody marki Volvo, posortuj je po pojemności skokowej, a następnie wyświetl tylko trzy kolumny: Markę, Cenę.w.PLN i Kolor.

- Wybierz tylko te kolumny, których nazwa kończy się na  $a$  (wyrażenie regularne opisujące  $a$  na końcu nazwy to  $a\$$ , więcej o wyrażeniach regularnych [http://pl.wikipedia.org/wiki/Wyra%C5%BCenie\\_regul](http://pl.wikipedia.org/wiki/Wyra%C5%BCenie_regul)

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Wyznaczanie nowych zmiennych danych [dplyr / mutate]

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 21*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Ptaki vs. koty](#)
- [Wyliczanie nowych zmiennych](#)
- [Nadpisywanie zmiennych](#)
- [Auta](#)
- [Wyliczanie jednej nowej zmiennej](#)
- [Wyliczanie nowych zmiennych](#)
- [Wyliczanie nowych zmiennych](#)
- [Klimatyzacja i nie tylko](#)
- [Zadania](#)

## O czym jest ten odcinek

Bardzo często na podstawie istniejących danych tworzymy nowe. Mając prędkość w metrach na sekundę liczymy prędkość w kilometrach na godzinę. Mając wagę i wzrost liczymy BMI. Mając odległość w milach liczymy odległość w kilometrach i tak dalej.

W programie R na różne sposoby można dodać nową zmienną do zbioru danych. Z tych wszystkich sposobów wygodnym i szybkim jest z wykorzystaniem funkcji `mutate()` z pakietu `dplyr`.

W tym odcinku nauczymy się:

- jak do zbioru danych dodać nową kolumnę,
- jak nadpisać wartości w istniejącej kolumnie.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012` - oba dostępne w pakiecie `PogromcyDanych`.

## Ptaki vs. koty

Rozpocznijmy przykład dla tworzenia nowych zmiennych od zbioru danych o kotach i ptakach. Jest on na tyle mały, że wynik operacji będzie można przedstawić na ekranie.

```
library(PogromcyDanych)
```

## koty\_ptaki

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	z
## 4		Puma	80.00	1.7	70	z
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	
## 10	Orzel przedni		5.00	0.9	160	
## 11	Sokol wedrowny		0.70	0.5	110	
## 12	Sokol norweski		2.00	0.7	100	
## 13	Albatros		4.00	0.8	120	Pe
##	waga	Kategoria	habitat	napis		
## 1	(100,1e+03]		Azja			
## 2	(100,1e+03]		Afryka			
## 3	(10,100]		Ameryka			
## 4	(10,100]		Ameryka			
## 5	(10,100]		Azja			
## 6	(10,100]		Afryka			
## 7	(10,100]		Azja			
## 8	(0,1]		Euroazja			
## 9	(100,1e+03]		Afryka			
## 10	(1,10]		Polnoc			
## 11	(0,1]		Polnoc			
## 12	(1,10]		Polnoc			
## 13	(1,10]		Poludnie			

## Wyliczanie nowych zmiennych

W zbiorze danych `koty_ptaki` jest zmienna `predkosc` opisująca prędkość w kilometrach na godzinę.

Dodajmy dwie nowe zmienne. W jednej przeliczymy prędkość z kilometrów na godzinę na prędkość w milach na godzinne, w drugiej przeliczymy prędkość na liczbę długości na sekundę (długość zwierząt jest w metrach). Nową zmienną można dodać funkcją `mutate()`. Pierwszym argumentem jest ramka danych, a kolejne to deklaracje nowych zmiennych w postaci `nazwa.zmiennej = wzór.na.nową.zmienną`.

W przetwarzaniu danych wykorzystamy operator `%>%` omówiony w 20 odcinku.

```
library(dplyr)
koty_ptaki %>%
  mutate(predkosc.mph = round(predkosc * 0.621371),
         dlugosci.na.sek = round(predkosc / 3.6))
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	z
## 4		Puma	80.00	1.7	70	z
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	En
## 9		Strus	150.00	2.5	70	
## 10	Orzel przedni		5.00	0.9	160	
## 11	Sokol wedrowny		0.70	0.5	110	

##	12	Sokol norweski	2.00	0.7	100
##	13	Albatros	4.00	0.8	120
##		wagaKategoria	habitat_napis	predkosc.mph	
##	1	(100,1e+03]	Azja		37
##	2	(100,1e+03]	Afryka		50
##	3	(10,100]	Ameryka		56
##	4	(10,100]	Ameryka		43
##	5	(10,100]	Azja		53
##	6	(10,100]	Afryka		71
##	7	(10,100]	Azja		40
##	8	(0,1]	Euroazja		106
##	9	(100,1e+03]	Afryka		43
##	10	(1,10]	Polnoc		99
##	11	(0,1]	Polnoc		68
##	12	(1,10]	Polnoc		62
##	13	(1,10]	Poludnie		75

## Nadpisywanie zmiennych

Funkcję `mutate()` można też wykorzystać do zmiany wartości w zmiennej, już obecnej w zbiorze danych.

W tym celu wystarczy, że nową wartość przypiszemy do nazwy istniejącej kolumny.

W przykładzie poniżej przeliczamy długość na cale i zaokrąglamy wynik do całkowitych liczb.

```
library(dplyr)
koty_ptaki %>%
  mutate(dlugosc = round(dlugosc * 100 / 2.54))
```



##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	98	60	
## 2		Lew	200.00	79	80	
## 3		Jaguar	100.00	67	90	2
## 4		Puma	80.00	67	70	2
## 5		Leopard	70.00	55	85	
## 6		Gepard	60.00	55	115	
## 7		Irbis	50.00	51	65	
## 8		Jerzyk	0.05	8	170	Eu
## 9		Strus	150.00	98	70	
## 10	Orzel przedni		5.00	35	160	
## 11	Sokol wedrowny		0.70	20	110	
## 12	Sokol norweski		2.00	28	100	
## 13	Albatros		4.00	31	120	Pe
##		wagaKategoria	habitat_napis			
## 1		(100,1e+03]	Azja			
## 2		(100,1e+03]	Afryka			
## 3		(10,100]	Ameryka			
## 4		(10,100]	Ameryka			
## 5		(10,100]	Azja			
## 6		(10,100]	Afryka			
## 7		(10,100]	Azja			
## 8		(0,1]	Euroazja			
## 9		(100,1e+03]	Afryka			
## 10		(1,10]	Polnoc			
## 11		(0,1]	Polnoc			
## 12		(1,10]	Polnoc			
## 13		(1,10]	Poludnie			

# Auta

Operacje tworzenia nowych zmiennych przećwiczmy też

na znacznie większym i złożonym zbiorze danych. Mianowicie na zbiorze danych o cenach ofertowych używanych samochodów.

Zbiór danych `auta2012` jest dostępny po wczytaniu pakietu `PogromcyDanych`. Poniżej przedstawiamy 6 pierwszych wierszy z tego zbioru danych. Szczegółowy opis tego zbioru znajduje się na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

```
head(auta2012)

## Source: local data frame [6 x 28]
##
##   Cena Waluta Cena.w.PLN Brutto.netto KM
## 1  49900    PLN      49900      brutto 140
## 2  88000    PLN      88000      brutto 156
## 3  86000    PLN      86000      brutto 150
## 4  25900    PLN      25900      brutto 163
## 5  55900    PLN      55900      netto  NA
## 6  45900    PLN      45900      netto 150
## Variables not shown: Wersja (fctr), Liczba...
##   (dbl), Przebieg.w.km (dbl), Rodzaj.paliwa
##   Kolor (fctr), Kraj.aktualnej.rejestracji
##   (fctr), Pojazd.uszkodzony (fctr), Skrzynia
##   Status.pojazdu.sprawadzonego (fctr), Wypos
##   Rodzaj.paliwa.posortowany (fctr), Kolor_na
##   Wyposazenie.dodatkowe_napis (chr), czy_me
##   (lgl), szyby (lgl), MarkaModel (chr)
```

# Wyliczanie jednej nowej zmiennej

Policzmy najpierw wiek auta. Dane były zbierane w roku 2012, więc wiek auta to różnica pomiędzy rokiem 2012 plus 1, a rokiem produkcji.

```
autaZWiekem <- auta2012 %>%  
  mutate(Wiek.auta = 2013 - Rok.produkcji)
```

Następnie wyświetlimy pierwsze 6 wierszy i dwie kolumny (wiek i rok produkcji).

```
autaZWiekem %>%  
  select(Wiek.auta, Rok.produkcji) %>%  
  head()
```

```
## Source: local data frame [6 x 2]
```

```
##  
##   Wiek.auta Rok.produkcji  
## 1         5         2008  
## 2         5         2008  
## 3         4         2009  
## 4        10         2003  
## 5         6         2007  
## 6         9         2004
```

---

## Wyliczanie nowych zmiennych

Zauważmy, że część cen jest netto, a część brutto. Przeliczymy wszystkie ceny na brutto dodając 23% tam gdzie podana jest cena netto.

Wykorzystamy w tym celu funkcję `ifelse()`, która jako wynik zwraca drugi lub trzeci argument w zależności od tego czy pierwszy argument jest prawdziwy czy nie.

```
autaZCenaBrutto <- auta2012 %>%  
  mutate(Cena.brutto = Cena.w.PLN * ifelse(Bru
```

Wyświetlmy pierwsze sześć wierszy i wybrane trzy kolumny.

```
autaZCenaBrutto %>%  
  select(Cena.brutto, Brutto.netto, Cena.w.PLN)  
head()
```

```
## Source: local data frame [6 x 3]  
##  
##   Cena.brutto Brutto.netto Cena.w.PLN  
## 1      49900      brutto      49900  
## 2      88000      brutto      88000  
## 3      86000      brutto      86000  
## 4      25900      brutto      25900  
## 5      68757      netto      55900  
## 6      56457      netto      45900
```

## Wyliczanie nowych zmiennych

W jednym wykonaniu funkcji `mutate()` można podać kilka transformacji.

Na poniższym przykładzie w jednym kroku wykonujemy

obie powyżej przedstawione transformacje.

```
autaZWiekciemIBrutto <- auta2012 %>%  
  mutate(Wiek.auta = 2013 - Rok.produkcji,  
         Cena.brutto = Cena.w.PLN * ifelse(Brutto.netto < 50000, 1.2, 1.1))
```

Wyświetlmy wybrane kolumny.

```
autaZWiekciemIBrutto %>%  
  select(Cena.brutto, Brutto.netto, Cena.w.PLN,  
         head())
```

```
## Source: local data frame [6 x 5]  
##  
##   Cena.brutto Brutto.netto Cena.w.PLN Wiek.auta  
## 1         49900         brutto      49900      1  
## 2         88000         brutto      88000      2  
## 3         86000         brutto      86000      3  
## 4         25900         brutto      25900      4  
## 5         68757          netto      55900      5  
## 6         56457          netto      45900      6
```

## Klimatyzacja i nie tylko

Wykorzystajmy funkcję `mutate()` aby dodać kolumny określające czy dane auto ma klimatyzację, centralny zamek czy autoalarm.

Aby sprawdzić czy w kolumnie `Wyposazenie.dodatkowe` występuje określony element użyjemy funkcji `grepl()`

```

autaZWyposazeniem <- auta2012 %>%
  mutate(Autoalarm = grepl(pattern = "autoalarm", Centralny.zamek = grepl(pattern = "centralny.zamek", Klimatyzacja = grepl(pattern = "klimatyzacja")

```

Wyświetlmy wybrane kolumny.

```

autaZWyposazeniem %>%
  select(Autoalarm, Centralny.zamek, Klimatyzacja)
head()

```

```

## Source: local data frame [6 x 3]
##
##   Autoalarm Centralny.zamek Klimatyzacja
## 1      TRUE              TRUE          TRUE
## 2     FALSE              TRUE          TRUE
## 3      TRUE              TRUE          TRUE
## 4      TRUE              TRUE          TRUE
## 5     FALSE              TRUE         FALSE
## 6     FALSE              TRUE          TRUE

```

## Zadania

- Dla zbioru danych `koty_ptaki` dodajmy nową zmienną. Z fizyki wiemy, że pęd to prędkość razy masa. Policz maksymalne pędy dla każdego gatunku oraz uporządkuj wiersze w kolejności od tych zdolnych do uzyskiwania najwyższego pędu.
- Policz średni przebieg na rok, dzieląc przebieg przez wieka auta.

- Poza koniem mechanicznym inną ciekawą jednostką mocy jest koń parowy (jednostka HP). Jeden koń mechaniczny to 0.9863 konia parowego. Utwórz nową zmienną, która przedstawi moc aut w koniach parowych (btw: Wikipedia zna jeszcze kilka innych ciekawych jednostek mocy).

Przykładowe odpowiedzi znajdują się na stronie

[http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)

# Agregaty danych [dplyr / summarise]

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 22*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Ptaki vs. koty](#)
- [Agregaty](#)
- [Agregaty szerzej](#)
- [Auta](#)
- [Agregaty](#)
- [Agregaty](#)
- [Zadania](#)

## O czym jest ten odcinek

Ogląd całych danych jest możliwy, tylko gdy dane nie są zbyt duże. Mieszczą się na jednym lub kilku ekranach. Dla większych zbiorów danych konieczne jest agregowanie informacji z poziomu pojedynczych wierszy na poziom



grup.

Aby efektywnie przedstawiać agregaty w grupach potrzebujemy funkcji liczącej agregaty oraz funkcji definiującej grupy. Do agregacji wygodnie wykorzystać funkcję `summarise()` z pakietu `dplyr`, którą przedstawimy w tym odcinku.

W tym odcinku nauczymy się:

- jak dla całego zbioru danych wyznaczać agregaty,
- jakie przykładowe agregaty można wyznaczać.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012`. Oba dostępne w pakiecie `PogromcyDanych`.

---

## Ptaki vs. koty

Rozpocznijmy od przykładu dla małego zbioru danych. Widząc wszystkie wiersze łatwiej będzie nam zauważyć zależność pomiędzy oryginalnymi danymi a agregatami. Ten zbiór danych jest dostarczany razem z pakietem `PogromcyDanych`.

```
library(PogromcyDanych)
```

koty\_ptaki

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	A
## 4		Puma	80.00	1.7	70	A
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	
## 10	Orzel	przedni	5.00	0.9	160	
## 11	Sokol	wedrowny	0.70	0.5	110	
## 12	Sokol	norweski	2.00	0.7	100	
## 13		Albatros	4.00	0.8	120	Pe
##	waga		Kategoria	habitat	napis	
## 1	(100,1e+03]		Azja			
## 2	(100,1e+03]		Afryka			
## 3	(10,100]		Ameryka			
## 4	(10,100]		Ameryka			
## 5	(10,100]		Azja			
## 6	(10,100]		Afryka			
## 7	(10,100]		Azja			
## 8	(0,1]		Euroazja			
## 9	(100,1e+03]		Afryka			
## 10	(1,10]		Polnoc			
## 11	(0,1]		Polnoc			
## 12	(1,10]		Polnoc			
## 13	(1,10]		Poludnie			

Do wyznaczania agregatów / podsumowań wykorzystamy funkcję `summarise()`. Jako pierwszy argument przyjmuje ona zbiór danych (w poniższym przykładzie przesłany za pomocą operatora `%>%`), kolejne argumenty to deklaracje, jak liczony ma być agregat oraz jak ma się nazywać odpowiadająca mu kolumna.

W pierwszej linii wczytujemy potrzebny pakiet `dplyr` a następnie liczymy następujące statystyki: minimalne i maksymalne prędkości, maksymalną i średnią wagę oraz coś ciut trudniejszego, maksymalną długość nazwy gatunku.

Każdą z tych wartości moglibyśmy policzyć również bez funkcji `summarise()`, zalety korzystania z niej w pełni objawia się w kolejnym odcinku.

```
library(dplyr)

koty_ptaki %>%
  summarise(najszybszy = max(predkosc),
            najwolniejszy = min(predkosc),
            najciezszy = max(waga),
            srednia.waga = mean(waga),
            najdluzsza.nazwa = max(nchar(as.cha
```

```
##      najszybszy najwolniejszy najciezszy sredn:
## 1           170              60         300      78
```

---

# Agregaty szerzej

Tworząc agregaty możemy je opierać na więcej niż jednej zmiennej. W poniższych przykładach liczymy średni i medianowy pęd oraz maksymalną prędkość mierzoną w liczbach długości ciała na sekundę. W każdym przykładzie statystykę liczymy na dwóch zmiennych.

```
koty_ptaki %>%
  summarise(sredni.ped = mean(waga*predkosc),
            medianowy.ped = median(waga*predkosc),
            predkosc.w.dlugosciach.na.sek = max(waga*predkosc))

##      sredni.ped medianowy.ped predkosc.w.dlugosciach.na.sek
## 1      5905.038           5600
```

Tworząc agregaty możemy wykorzystywać agregaty policzone w poprzednim kroku. W poniższym przykładzie liczymy średnią wagę a później przeliczamy ją na funty.

```
koty_ptaki %>%
  summarise(srednia.waga = mean(waga),
            srednia.waga.w.funtach = srednia.waga*2.20462)

##      srednia.waga srednia.waga.w.funtach
## 1      78.59615      173.5014
```

---

## Auta

Pokażmy jeszcze przykład działania agregatów na większym zbiorze danych o cenach ofertowych samochodów. Będziemy pracować na kilku kolumnach. Poniżej wyświetlamy tylko markę, model cenę i przebieg dla pierwszych 6 wierszy ze zbioru danych `auta2012`.

```
auta2012 %>%
  select(Marka, Model, Cena.w.PLN, Przebieg.w.l)
head()
```

```
## Source: local data frame [6 x 4]
##
##      Marka      Model  Cena.w.PLN Przebieg.w.l
## 1      Kia    Carens    49900
## 2 Mitsubishi Outlander    88000
## 3 Chevrolet  Captiva    86000
## 4      Volvo      S80    25900
## 5 Mercedes-Benz Sprinter    55900
## 6 Mercedes-Benz  Viano    45900
```

# Agregaty

Zbiór danych `auta2012` zawiera szczegółowe informacje o parametrach każdej oferty sprzedaży. Często z takich zbiorów danych interesują nas pewne statystyki agregujące informacje ze zbioru danych.

Przykładem może być średnia cena, rozrzut ceny, przykładowo mierzony przez odchylenie standardowe

ceny, połówkowy przebieg czy jakiego przebiegu nie przekracza połowa aut.

Takie agregaty możemy wyznaczyć funkcją `summarise()`. Jako pierwszy argument podajemy zbiór danych a jako kolejne wskazujemy statystyki / agregaty, które chcemy wyznaczyć z tego zbioru danych.

```
auta2012 %>%  
  summarise(sredniaCena = mean(Cena.w.PLN),  
            sdCena = sd(Cena.w.PLN),  
            medianaPrzebiegu = median(Przebieg))  
  
## Source: local data frame [1 x 3]  
##  
##   sredniaCena    sdCena medianaPrzebiegu  
## 1      35755.11  70399.67           140000
```

Funkcja `mean()` wyznacza średnią, funkcja `var()` wyznacza wariancję a funkcja `sd()` wyznacza odchylenie standardowe, funkcja `median()` wyznacza medianę obserwacji w grupie.

Nie wszystkie wartości są podane dla wszystkich zmiennych. Jeżeli jakaś wartość nie została podana, to w zbiorze danych jest ona zakodowana jako NA (ang. Not Available) - brakująca wartość.

Jeżeli w zmiennej są wartości brakujące, to argument `na.rm=TRUE` oznacza, że wynik ma być wyznaczony po

usunięciu wartości brakujących.

---

## Agregaty

Definicja agregatu może być bardziej rozbudowana niż jedna instrukcja. Przykładowo aby policzyć liczbę lub procent aut z klimatyzacją w pierwszym kroku możemy użyć funkcji `grepl()`, aby sprawdzić czy auto w danej ofercie ma klimatyzację, a następnie zsumować wyniki tej funkcji, by otrzymać liczbę aut z klimatyzacją.

Licząc procent aut z automatyczną skrzynią biegów w pierwszym kroku możemy stworzyć zmienną logiczną określającą czy zmienna `Skrzynia.biegow` przyjmuje wartość `automatyczna`, a następnie policzyć procent za pomocą operacji `100*mean()`

Tworząc agregaty wygodnie jest korzystać z funkcji `n()`, której wynikiem jest liczba wierszy w zbiorze danych / grupie.

```
auta2012 %>%
  summarise(liczba.aut.z.klimatyzacja = sum(grepl("klimatyzacja",
  procent.aut.z.klimatyzacja = 100*mean(liczba.aut.z.klimatyzacja)
  procent.aut.z.automatem = 100*mean(liczba.aut = n())
```

```
## Source: local data frame [1 x 4]
```

```
##  
##      liczba.aut.z.klimatyzacja procent.aut.z.k.  
## 1                                162960  
## Variables not shown: procent.aut.z.automater
```

---

## Zadania

- Policz sumaryczny przebieg wszystkich samochodów. Następnie policz ile razy okrążono kulę ziemską uwzględniając te wszystkie przebiegi.
- Wybierz tylko samochody marki ‘Rolls-Royce’ i policz ich średni przebieg oraz średnią cenę.

Przykładowe odpowiedzi znajdują się na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)



# Grupowanie danych i analiz [dplyr / group\_by]

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 23*

*pogromcy danych*

- [O czym jest ten odcinek](#)
- [Ptaki vs. koty](#)
- [Ptaki vs. koty](#)
- [Auta](#)
- [Grupowanie](#)
- [Sortowanie](#)
- [Grupowanie po dwóch zmiennych](#)
- [Grupowanie i filtrowanie](#)
- [Zadania](#)
- [Co dalej](#)

## O czym jest ten odcinek

W poprzednim odcinku pokazaliśmy jak liczyć na zbiorze danych statystyki / agregaty. Użyteczność tej opcji

znacząco rośnie gdy liczymy te agregaty w grupach. Dzięki temu możemy później łatwo te grupy porównać. Do określania grup wykorzystamy funkcję `group_by()` z pakietu `dplyr`.

W tym odcinku nauczymy się:

- jak określać jedną lub więcej zmiennych grupujących,
- jak liczyć agregaty w grupach.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, mały zbiór danych, to `koty_ptaki` a drugi, znacznie większy, to `auta2012`. Oba zbiory są dostępne w pakiecie `PogromcyDanych`.

---

## Ptaki vs. koty

Rozpocznijmy od przykładu dla danych o kotach i ptakach. Jest on na tyle mały, że wynik operacji będzie można przedstawić na ekranie.

Zmienna `druzyna` nadaje się świetnie na zmienną grupującą.

```
library(PogromcyDanych)
koty_ptaki
```

##		gatunek	waga	dlugosc	predkosc	
## 1		Tygrys	300.00	2.5	60	
## 2		Lew	200.00	2.0	80	
## 3		Jaguar	100.00	1.7	90	2
## 4		Puma	80.00	1.7	70	2
## 5		Leopard	70.00	1.4	85	
## 6		Gepard	60.00	1.4	115	
## 7		Irbis	50.00	1.3	65	
## 8		Jerzyk	0.05	0.2	170	Eu
## 9		Strus	150.00	2.5	70	
## 10	Orzel przedni		5.00	0.9	160	
## 11	Sokol wedrowny		0.70	0.5	110	
## 12	Sokol norweski		2.00	0.7	100	
## 13	Albatros		4.00	0.8	120	Pe
##		wagaKategoria	habitat_napis			
## 1		(100,1e+03]	Azja			
## 2		(100,1e+03]	Afryka			
## 3		(10,100]	Ameryka			
## 4		(10,100]	Ameryka			
## 5		(10,100]	Azja			
## 6		(10,100]	Afryka			
## 7		(10,100]	Azja			
## 8		(0,1]	Euroazja			
## 9		(100,1e+03]	Afryka			
## 10		(1,10]	Polnoc			
## 11		(0,1]	Polnoc			
## 12		(1,10]	Polnoc			
## 13		(1,10]	Poludnie			

# Ptaki vs. koty

Funkcja `group_by()` jako pierwszy argument przyjmuje zbiór danych, a jako kolejne zmienne grupujące (jedną lub

więcej). Sama funkcja `group_by()` nie powoduje wyliczenia nowych wartości, a jedynie zaznacza, które zmienne mają być użyte do grupowania, przez co użycie kolejnych funkcji (takich jak np. `summarise()`) będzie miało zmienione działanie.

Zobaczmy jak wygląda użycie funkcji `group_by()` do porównania maksymalnej wagi, prędkości i żywotności w grupie ptaków i kotów.

Wynikiem jest ramka danych, której kolumny to w pierwszej kolejności zmienne grupujące, a następnie wyniki agregatów w grupach.

```
library(dplyr)

koty_ptaki %>%
  group_by(druzyna) %>%
  summarise(najszybszy = max(predkosc),
            najciezszy = max(waga),
            najzywotniejszy = max(zywotnosc),
            liczba = n())
```

```
## Source: local data frame [2 x 5]
##
##   druzyna najszybszy najciezszy najzywotniejszy
## 1      Kot         115         300
## 2     Ptak         170         150
```

Aby przedstawić więcej możliwości, które daje operacja grupowania wykorzystamy zbiór danych `auta2012`.

Poniżej przedstawiamy 6 pierwszych wierszy i wybrane kolumny z tego zbioru danych. Szczegółowy opis tego zbioru znajduje się na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzai](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzai)

```
auta2012 %>%
  select(Cena.w.PLN, Marka, Przebieg.w.km, Rodzaj)
head()
```

```
## Source: local data frame [6 x 5]
##
##   Cena.w.PLN      Marka Przebieg.w.km Rodzaj
## 1      49900      Kia      41000      Diesel
## 2      88000  Mitsubishi      46500      Diesel
## 3      86000  Chevrolet      8000       Benzyna
## 4      25900      Volvo     200000      Diesel
## 5      55900 Mercedes-Benz     169400      Diesel
## 6      45900 Mercedes-Benz     141100      Diesel
```

# Grupowanie

Przykłady dla danych o cenach ofertowych aut rozpoczniemy od porównania cen i przebiegów samochodów napędzanych różnymi paliwami.

Aby sensownie porównywać różne rodzaje paliwa

ograniczymy się do jednej marki samochodów i jednego rocznika. Na poniższym przykładzie wybieramy tylko 5-letnie Volkswageny, a następnie dla aut napędzanych benzyną, olejem czy gazem liczymy medianę ceny, medianę przebiegu oraz liczbę aut w każdej grupie.

```
auta2012 %>%  
  filter(Marka == "Volkswagen", Rok.produkcji = 2007, 2008, 2009, 2010, 2011)  
  group_by(Rodzaj.paliwa) %>%  
  summarise(medianaCeny = median(Cena.w.PLN, na.rm=T),  
            medianaPrzebieg = median(Przebieg.km, na.rm=T),  
            liczba = n())
```

```
## Source: local data frame [3 x 4]  
##  
##   Rodzaj.paliwa medianaCeny medianaPrzebieg  
## 1              33550.0         67000  
## 2              B      34048.9         95000  
## 3              H      38900.0        145000
```

Najliczniejszą grupą aut są samochody napędzane olejem napędowym, mają one przeszło dwukrotnie większy przebieg niż napędzane benzyną (patrzac na mediany), a mimo to są droższe o około 15%.

---

## Sortowanie

Na agregatach możemy pracować tak jak na zwykłych ramkach danych. Możemy przekazać je dalej do funkcji,

które sortują, wybierają lub zmieniają zmienne.

W poniższym przykładzie sortujemy agregaty po liczbie obserwacji.

```
auta2012 %>%
  filter(Marka == "Volkswagen", Rok.produkcji = 2012) %>%
  group_by(Rodzaj.paliwa) %>%
  summarise(
    medianaCeny = median(Cena.w.PLN, na.rm=T),
    medianaPrzebieg = median(Przebieg.km, na.rm=T),
    liczba = n()) %>%
  arrange(liczba)
```

```
## Source: local data frame [3 x 4]
```

	Rodzaj.paliwa	medianaCeny	medianaPrzebieg
1	B	34048.9	95000
2		33550.0	67000
3	H	38900.0	145000

## Grupowanie po dwóch zmiennych

Grupować można po kilku zmiennych, w tym przypadku agregaty liczone są w każdym podzbiorze zmiennych.

Zobaczmy jak będą wyglądały przebiegi i ceny gdy auta podzielimy dodatkowo na modele. W grupie pięcioletnich Volkswagenów znaleźć można 19 różnych modeli.

Poniższa lista instrukcji jest praktycznie identyczna z

poprzednią, różnica polega na tym, że grupujemy po dwóch zmiennych, Modelu i Rodzaju paliwa.

```
auta2012 %>%
  filter(Rok.produkcji == 2007, Marka == "Volks")
  group_by(Model, Rodzaj.paliwa) %>%
  summarise(medianaCeny = median(Cena.w.PLN, na.rm=T),
            medianaPrzebieg = median(Przebieg.w.km, na.rm=T),
            liczba = n())

## Source: local data frame [35 x 5]
## Groups: Model
##
##      Model Rodzaj.paliwa medianaCeny medianaPrzebieg liczba
## 1 Beetle                39000.0          100000.0      1
## 2 Caddy                  27900.0          100000.0      1
## 3 Caddy                   H      30813.0          100000.0      1
## 4 Caravelle              H      65900.0          100000.0      1
## 5 Eos                    53445.0          100000.0      1
## 6 Eos                     H      64900.0          100000.0      1
## 7 Fox                    15227.9          100000.0      1
## 8 Fox                     H      17000.0          100000.0      1
## 9 Golf                    35900.0          100000.0      1
## 10 Golf                   B      31650.0          100000.0      1
## .. ...
```

## Grupowanie i filtrowanie

W wyniku z poprzedniego działania wiele grup miało małą liczebność, przez co wyniki dla nich były dosyć przypadkowe, a to utrudnia porównywania.



Dodanie funkcji `filter()` na koniec umożliwia pozostawienie tylko tych grup, w których liczebność przekracza 10 obserwacji.

Warto zwrócić uwagę, że w poniższym przykładzie funkcja `filter()` występuje dwukrotnie. W pierwszym przypadku filtruje wiersze w oryginalnym zbiorze danych, a w drugim filtruje grupy o zbyt małej liczebności.

```
auta2012 %>%
  filter(Rok.produkcji == 2007, Marka == "Volks")
group_by(Model, Rodzaj.paliwa) %>%
  summarise(medianaCeny = median(Cena.w.PLN, na
                                medianaPrzebieg = median(Przebieg.w.km, na
                                liczba = n())) %>%
  filter(liczba > 10)
```

```
## Source: local data frame [17 x 5]
```

```
## Groups: Model
```

```
##
```

##	Model	Rodzaj.paliwa	medianaCeny	medianaPrzebieg
## 1	Caddy	H	30813.00	
## 2	Fox		15227.90	
## 3	Golf		35900.00	
## 4	Golf	H	33900.00	
## 5	Golf Plus		26948.25	
## 6	Golf Plus	H	35900.00	
## 7	Jetta		36700.00	
## 8	Jetta	H	37250.00	
## 9	Multivan	H	92222.90	
## 10	Passat		39200.00	
## 11	Passat	H	41000.00	
## 12	Polo		23990.00	

##	13	Polo	H	24900.00
##	14	Sharan	H	45745.00
##	15	Touareg	H	103450.00
##	16	Touran	H	40900.00
##	17	Transporter	H	49450.00

## Zadania

- Wybierz pięcioletnie Golfy i sprawdź czy średnia cena zależy od kraju aktualnej rejestracji. W tym celu pogrupuj po zmiennej `Kraj.aktualnej.rejestracji` i w każdej grupie policz średnią.
- Wybierz tylko Peugeoty 206 i policz średnią cenę w zależności od roku produkcji. Posortuj te grupy po roku produkcji.

Przykładowe odpowiedzi znajdują się na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzania](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzania)

## Co dalej

Pakiet `dplyr` zupełnie zmienił sposób przetwarzania danych w R. Przetwarzanie stało się prostsze do opisanego, 80% niezbędnej pracy można wykonać łatwiej i szybciej.

Świetnym uzupełnieniem do wyników z odcinków o tym

pakiecie będzie graficzna dwustronicowa ściągawka, prezentująca możliwości tego pakietu. Ta ściągawka dostępna jest na stronie <http://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

Więcej takich skrótowych ściągawek jest dostępnych na stronie firmy RStudio

<http://www.rstudio.com/resources/cheatsheets/>

Krótką prezentacją najważniejszych funkcji tego pakietu na przykładzie danych o lotach znaleźć można na stronie warsztatów twórcy dplyr'a

<http://cran.rstudio.com/web/packages/dplyr/vignettes/intro>

# Formatowanie i przekształcanie danych [tidyr]

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 24*  
*pogRomcy danych*

- [O czym jest ten odcinek](#)
- [Dane o pracy nerki](#)
- [Dane o pracy nerki - postać wąska](#)
- [Dane o pracy nerki - postać wąska](#)
- [Dane z EuroStatu](#)
- [Z postaci wąskiej do szerokiej](#)
- [Z postaci wąskiej do szerokiej](#)
- [Z szerokiej do wąskiej](#)
- [Sklejanie kolumn](#)
- [Rozszczepianie kolumn](#)
- [Rozszczepianie kolumn](#)
- [Zadania](#)

# O czym jest ten odcinek

Dane najczęściej są przedstawiane w postaci tabelarycznej. Jednak mogą być w tej tabeli różnie sformatowane. Wyróżnia się między innymi reprezentacje szeroką danych, wąską danych i reprezentacje mieszane.

Po co taka różnorodność? Otóż w zależności od tego co z danymi chcemy zrobić czasem lepiej je mieć w takiej czy innej postaci. Pakiet `tidyr` udostępnia funkcje, aby szybko i wygodnie przechodzić z jednego sposobu reprezentacji na drugi.

W tym odcinku nauczymy się:

- jak przejść z wąskiej postaci na szeroką,
- jak przejść z szerokiej postaci do wąskiej.

Do ilustracji tych zagadnień wykorzystamy dwa zbiory danych. Pierwszy, to zbiór danych `kidney` z pakietu `PBImisc`, pozwoli nam na ilustracje przejścia z formatu szerokiego na wąski. Drugi zbiór danych to dane z Eurostatu, które pobierzemy z użyciem pakietu `PogromcyDanych`. Pozwolą one zilustrować przejście z reprezentacji wąskiej danych na szeroką.

---

# Dane o pracy nerki

Aby zilustrować czym jest szeroka reprezentacja danych i jak przejść z niej na wąską reprezentację wykorzystamy zbiór danych `kidney` z pakietu `PBImisc`. Ten pakiet nie jest dostępny wraz z bazową wersją R. Należy go doinstalować funkcją `install.packages()` (informacja o tym jak to zrobić jest w odcinku 2). Po instalacji włączymy ten pakiet. Do zbioru danych `kidney` dodajemy kolumnę z `id` pacjenta.

```
library(PBImisc)
kidney$id <- rownames(kidney)
```

Następnie ze zbioru `kidney` wybierzemy kilka interesujących nas kolumn. Każdy wiersz w tym zbiorze danych opisuje jednego pacjenta. W kolumnie `therapy` znajduje się zmienna jakościowa o trzech poziomach, a w kolumnach `MDRD7 ... MDRD60` znajdują się pomiary sprawności nerki (mierzone współczynnikiem MDRD) w dniach 7, 30, oraz miesiącach 3, 6, 12, 24, 36 i 60 po przeszczepie.

Do wyboru interesujących nas zmiennych wygodnie jest użyć funkcji `select()` z pakietu `dplyr`.

Taką reprezentację danych, w której kolejne pomiary MDRD opisane są przez kolejne kolumny, nazywa się

reprezentacją szeroką. Im więcej punktów pomiarowych dla MDRD tym więcej kolumn w zbiorze danych.

```
library(dplyr)
kidney_wybrane <- kidney %>%
  dplyr::select(id, therapy, MDRD7:MDRD60)

head(kidney_wybrane)
```

##		id	therapy	MDRD7	MDRD30	MDRD3	MDRD6	MDRD12
##	1	1	tc	46	71	65.0	71	65.0
##	2	2	cm	44	58	20.0	78	65.0
##	3	3	cm	6	36	37.8	42	45.0
##	4	4	tc	8	39	55.7	52	60.0
##	5	5	ca	36	79	64.2	64	64.0
##	6	6	cm	9	43	58.0	49	45.0

Funkcja `select()` występuje w różnych pakietach, aby mieć pewność, że uruchomiona będzie funkcja z pakietu `dplyr`, dodane jest wskazanie pakietu z użyciem operatora `::`. Zapis `dplyr::select()` jawnie wskazuje, że chodzi o funkcję `select()` z pakietu `dplyr`.

---

## Dane o pracy nerki - postać wąska

Czym w takim razie jest postać wąska? Łatwiej to wyjaśnić na przykładzie.

Użyjemy funkcji `gather()` z pakietu `tidyr`. Pierwszym

argumentem jest ramka danych w postaci szerokiej. Kolejne dwa argumenty (w poniższym przykładzie to `time` i `MDRD`) to nazwy zmiennych, które w nowym zbiorze danych będą opisywały klucze i wartości. Pozostałe argumenty to lista kolumn, które mają być przekształcone w reprezentację wąską (tutaj są to wszystkie `MDRD...`).

Wynikiem jest reprezentacja wąska, w której kolejne pomiary `MDRD` zostały przedstawione za pomocą różnych wierszy. W reprezentacji wąskiej w kolumnie `time` znajduje się nazwa kolumny z oryginalnego zbioru danych, a w kolumnie `MDRD` znajdują się wartości.

Zbiór danych `kidney_wybrane` miał 334 wiersze i 10 kolumn a zbiór danych `kidney_waska` ma 2672 wiersze i 4 kolumny. Każdy z 334 pacjentów jest opisany przez 8 wierszy, każdy wiersz dla innego punktu pomiarowego `MDRD`.

```
library(tidyr)

kidney_waska <- gather(kidney_wybrane, time, MDRD)
head(kidney_waska)
```

##		id	therapy	time	MDRD
##	1	1	tc	MDRD7	46
##	2	2	cm	MDRD7	44
##	3	3	cm	MDRD7	6
##	4	4	tc	MDRD7	8
##	5	5	ca	MDRD7	36
##	6	6	cm	MDRD7	9



---

# Dane o pracy nerki - postać wąska

Warto zauważyć, że zmienne nie wymienione w funkcji `gather()` (w tym przykładzie to zmienne `id` oraz `therapy`) zostaną pozostawione w ramce danych, nie będą skonwertowane na postać wąską.

```
kidney_waska %>% filter(id=="1")
```

##		id	therapy	time	MDRD
##	1	1	tc	MDRD7	46
##	2	1	tc	MDRD30	71
##	3	1	tc	MDRD3	65
##	4	1	tc	MDRD6	71
##	5	1	tc	MDRD12	65
##	6	1	tc	MDRD24	70
##	7	1	tc	MDRD36	76
##	8	1	tc	MDRD60	72

---

# Dane z EuroStatu

Do ilustracji transformacji z postaci wąskiej na szeroką wykorzystamy dane z Eurostatu. Dane w Eurostatie są przechowywane z kilkoma wymiarami, na takich danych wygodniej się pracuje, gdy są one w postaci wąskiej. Jednak gdy przychodzi o wyświetlania tych danych na

ekranie wygodniej jest je przekształcić na postać szeroką.

Do ilustracji tych funkcji wykorzystamy zbiór danych z Eurostatu o popularności transportu w różnych krajach. Te dane są w tabeli `tsdtr210` na serwerach Eurostat. Do pobrania tych danych użyjemy funkcji `getEurostatRCV()` z pakietu `SmarterPoland`.

```
library(SmarterPoland)
tsdtr210 <- getEurostatRCV("tsdtr210")
head(tsdtr210)
```

##		unit	vehicle	geo	time	va
##	PC_BUS_TOT_AT_1990	PC	BUS_TOT	AT	1990	
##	PC_BUS_TOT_BE_1990	PC	BUS_TOT	BE	1990	
##	PC_BUS_TOT_BG_1990	PC	BUS_TOT	BG	1990	
##	PC_BUS_TOT_CH_1990	PC	BUS_TOT	CH	1990	
##	PC_BUS_TOT_CY_1990	PC	BUS_TOT	CY	1990	
##	PC_BUS_TOT_CZ_1990	PC	BUS_TOT	CZ	1990	

Pobrane dane są w postaci wąskiej. Kolumna `geo` określa kraj, kolumna `time` określa rok, kolumna `vehicle` rodzaj transportu, a kolumna `value` popularność danego rodzaju transportu w określonym kraju, w określonym roku.

---

## Z postaci wąskiej do szerokiej

Aby przejść z postaci wąskiej do postaci szerokiej, można użyć funkcji `spread()`.

Jej pierwszym argumentem jest ramka z danymi w postaci wąskiej. Kolejne dwie zmienne określają kolumnę w której przechowywane są klucze i wartości.

Funkcja `spread()` konwertuje ramkę danych w ten sposób, że wartości drugiej kolumny wejściowego zbioru danych stają się nazwami kolumn wyjściowego zbioru danych.

W poniższym przykładzie zmienna `time` jest wskazana jako klucz, a więc wartości tej zmiennej staną się nazwami kolumn w nowym zbiorze danych. Zmienna `value` jest wskazana jako wartości i to ona wypełni nowe kolumny.

Pozostałe kolumny, w tym przypadku `vehicle` i `geo`, pozostają w niezmienionej postaci.

Wyświetlmy wiersze spełniające warunek `geo == "PL"`. Jak widzimy wynikowy zbiór danych `szeroka` ma 13 kolumn opisujących popularność różnych środków transportu w kolejnych latach.

```
szeroka <- spread(tsdtr210, time, value)
```

```
## wyświetlmy wiersze dla Polski
szeroka %>% filter(geo == "PL")
```

##	unit	vehicle	geo	1990	1991	1992	1993	1994
## 1	PC	BUS_TOT	PL	NA	NA	NA	NA	NA

##	2	PC	CAR	PL	41.3	NA	NA	NA	NA
##	3	PC	TRN	PL	30.6	NA	NA	NA	NA
##		2001	2002	2003	2004	2005	2006	2007	2008
##	1	14.7	13.5	13.5	13.1	12.0	10.6	9.6	8.4
##	2	74.7	77.0	77.6	78.9	80.7	82.5	83.6	85.5
##	3	10.6	9.5	8.8	8.0	7.3	6.9	6.8	6.2

## Z postaci wąskiej do szerokiej

Dane z Eurostatu miały więcej wymiarów i każdy z nich może być użyty do stworzenia nowych kolumn. Przykładowo w poniższym przykładzie do rozszerzenia ramki danych wykorzystamy zmienną `geo`.

```
szeroka2 <- spread(tsdtr210, geo, value)

## wyświetlmy wiersze dla roku 2010
szeroka2 %>% filter(time == "2010")

##      unit vehicle time      AT      BE      BG      CH      CY
## 1      PC  BUS_TOT 2010  10.3  12.2  16.4   5.1  18.1
## 2      PC      CAR 2010  78.7  80.3  80.0  77.3  81.9
## 3      PC      TRN 2010  11.0   7.4   3.6  17.6   NA
##      EU27 EU28      FI      FR      HR      HU      IE      IS
## 1   8.7   8.7   9.9   5.3  10.7  21.4  14.5  11.4  12.1
## 2  84.2  84.2  84.9  85.5  83.7  68.6  82.6  88.6  82.4
## 3   7.1   7.1   5.2   9.3   5.6   9.9   2.9   NA   5.1
##      NL      NO      PL      PT      RO      SE      SI      SK
## 1   3.7   6.8   6.4   6.5  12.9   7.2  10.8  15.5  38.1
## 2  87.2  88.4  88.4  89.1  81.3  83.4  86.8  77.8  59.1
## 3   9.1   4.8   5.2   4.4   5.9   9.4   2.5   6.7   2.4
```

# Z szerokiej do wąskiej

Proces przechodzenia z wąskiej postaci do szerokiej i z szerokiej do wąskiej jest odwracalny. Pokażmy jak z powrotem, przejść z postaci szerokiej do wąskiej, za pomocą funkcji `gather()`.

Poniżej pierwszy argument jest przekazany za pomocą operatora `%>%`, kolejne dwa określają nazwy nowych kolumn w których zapisane będą klucze i wartości. Trzeci argument to opis kolumn, które mają być zamienione na postać wąską.

W poniższym przykładzie funkcja `gather()` przekształca ramkę `szeroka` w ten sposób, że wszystkie kolumny poza `geo` i `vehicle` (a więc kolumny z nazwami lat) będą zakodowane przez parę klucz `rok` i wartość `wartosc`.

```
szeroka %>%  
  gather(rok, wartosc, -geo, -vehicle) %>%  
  tail()
```

##		vehicle	geo	rok	wartosc
##	2515	TRN	RO	2012	4.9
##	2516	TRN	SE	2012	9.1
##	2517	TRN	SI	2012	2.3
##	2518	TRN	SK	2012	7.1
##	2519	TRN	TR	2012	1.7
##	2520	TRN	UK	2012	8.2

Aby wyświetlić przykładowe 6 wierszy użyto tutaj funkcji `tail()` (wyświetla ostatnie sześć wierszy) ponieważ w pierwszych sześciu wierszach są wartości `NA`,

---

## Sklejanie kolumn

Zdarza się, że wartości z kilku kolumn chcemy skleić ze sobą w jedną kolumnę. Można to zrobić funkcją `unite()`.

Pierwszy argument tej funkcji to ramka danych. Drugi to nazwa kolumny, która zostanie utworzona przez połączenie kolumn, które są pozostałymi argumentami. Argument `sep` określa co ma rozdzielać wartości w połączonych kolumnach.

W przykładzie poniżej w zbiorze danych `tsdtr210` stworzymy nową kolumnę o nazwie `panstwo_rok`, której wartości powstaną przez połączenie wartości w kolumnach `geo` i `time` rozdzielając je znakiem `:`.

```
unite(tsdtr210, panstwo_rok, geo, time, sep=":")
head()
```

##		unit	vehicle	panstwo_rok
##	PC_BUS_TOT_AT_1990	PC	BUS_TOT	AT:1990
##	PC_BUS_TOT_BE_1990	PC	BUS_TOT	BE:1990
##	PC_BUS_TOT_BG_1990	PC	BUS_TOT	BG:1990
##	PC_BUS_TOT_CH_1990	PC	BUS_TOT	CH:1990

##	PC_BUS_TOT_CY_1990	PC	BUS_TOT	CY:1990
##	PC_BUS_TOT_CZ_1990	PC	BUS_TOT	CZ:1990

---

## Rozszczepianie kolumn

Operację odwrotną do sklejania, a więc rozcinanie kolumn można wykonać funkcją `separate()`.

Przedstawimy działanie tej funkcji na przykładzie sztucznego zbioru danych z dwoma kolumnami - datą i id.

```
df <- data.frame(daty = c("2004-01-01", "2012-04-15",  
                           "2006-10-29", "2010-03-03"),  
                 id = 1:4)  
df
```

##		daty	id
##	1	2004-01-01	1
##	2	2012-04-15	2
##	3	2006-10-29	3
##	4	2010-03-03	4

Pierwszym argumentem funkcji `separate()` jest ramka danych, którą chcemy przekształcić. Drugim argumentem jest nazwa kolumny, którą chcemy rozszczepić. Trzecim argumentem jest wektor napisów, które będą nazwami nowych - rozszczepionych kolumn. Czwarty argument to separator - znak, który rozszczepi kolumny. Poniżej przedstawiamy przykład, w którym kolumnę `daty` ze zbioru danych `df` rozszczepiamy na wartości rozdzielone

znakiem -. Poszczególne składowe będą stanowiły rok, miesiąc i dzień.

```
separate(df, daty, c("rok", "miesiac", "dzien"))
```

##		rok	miesiac	dzien	id
##	1	2004	01	01	1
##	2	2012	04	15	2
##	3	2006	10	29	3
##	4	2010	03	03	4

## Rozszczepianie kolumn

Nie tylko daty można rozszczepiać na części składowe.

Podobnie można przekształcić listy produktów lub wyposażenie. Na poniższym przykładzie pracujemy z ramką danych opisujących klasę, szkołę i miasto z którego pochodzi uczeń.

```
df <- data.frame(nr = c("PB", "RG", "AK", "NZ"),
                 klasa = c("Wroclaw, IV LO, IIIC",
                           "Warszawa, XIV LO, Ia",
                           "Wroclaw, III LO, IIb",
                           "Krakow, XXX LO, Ic"))
df
```

##		nr	klasa
##	1	PB Wroclaw, IV LO, IIIC	
##	2	RG Warszawa, XIV LO, Ia	
##	3	AK Wroclaw, III LO, IIb	
##	4	NZ Krakow, XXX LO, Ic	



Dane o szkole i klasie są wyciągane z użyciem funkcji `separate()`.

```
separate(df, klasa, c("miasto", "szkola", "kla
```

```
##      nr   miasto szkoła klasa
## 1 PB   Wrocław  IV LO   IIIc
## 2 RG   Warszawa XIV LO    Ia
## 3 AK   Wrocław  III LO   IIb
## 4 NZ    Kraków  XXX LO    Ic
```

---

## Zadania

- W bazie danych Eurostatu o nazwie `prc_ppp_ind` zbierane są informacje o średniej sile nabywczej. Odczytaj te dane funkcją `getEurostatRCV()`, a następnie zamień z postaci wąskiej do szerokiej.
- Ze zbioru danych `kidney` wybierz tylko kolumny `MDRD12`, `MDRD24`, `MDRD36`, `MDRD60` a następnie zamień je z postaci szerokiej do wąskiej.
- W zbiorze danych `kidney` niezgodności w antygenach AB i DR są opisane przez kolumny `discrepancy.AB` i `discrepancy.DR`. Zamień je w jedną kolumnę o nazwie `discrepancy` gdzie obie niezgodności są sklejone i separowane znakiem `_`.

Przykładowe odpowiedzi znajdują się na stronie [http://pogromcydanych.icm.edu.pl/materials/1\\_przetwarzaj](http://pogromcydanych.icm.edu.pl/materials/1_przetwarzaj)



# Zespół realizujący

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 28*

*pogRomcy danych*

*Opracowanie materiałów*

- Przemysław Biecek, ICM

*Dane wokół nas*

- Piotr Przybyła
- Bartosz Meglicki,
- zespół Na Straży Sondaży: Jakub Rutkowski,  
Zbigniew Marczewski, Przemysław Kopa

*Wsparcie merytoryczne i językowe*

- Paweł Chudzian
- Marcin Kosiński
- Katarzyna Łogwiniuk
- Krzysztof Trajkowski

## *Opracowanie wideo*

- Jarosław Skrzeczkowski, ICM.TV
- Łukasz Kałuża, ICM.TV

## *Strona graficzna*

- Podpunkt.pl

## *Lektor*

- Magda Karczewska

## *Opracowanie serwisu www*

- Paweł Dudek

## *System weryfikacji zadań*

- Cezary Chudzian

## *Wsparcie organizacyjne*

- Michał Bojanowski, ICM

## *Opracowanie zadań*

- Przemysław Biecek, ICM

# Co dalej?

*Przemysław Biecek @ Uniwersytet Warszawski*

*sezon 1 / odcinek 25*

*pogRomcy danych*

- [Książki](#)
  - [W języku polskim](#)
  - [W języku angielskim](#)
- [Dla użytkowników Excela](#)
- [Blogi](#)
  - [W języku polskim](#)
  - [W języku angielskim](#)
- [Spotkania użytkowników R](#)
  - [W Polsce](#)
  - [Na świecie](#)

Kurs, który właśnie Państwo ukończyli jest łagodnym wprowadzeniem do programu R. Potrafią Państwo teraz wczytać i przetwarzać dane oraz szukać informacji o dodatkowych funkcjonalnościach R.

Program R ma do zaoferowania znacznie więcej, niż zdążyliśmy zaznaczyć w tym prostym kursie.

W tym odcinku napiszemy gdzie szukać dalszych informacji o możliwościach R.

A aby sprawdzić swoje umiejętności zapraszamy do wzięcia udziału w zabawie w rozwiązanie 20 zadań związanych z R. Rozwiązanie dowolnych 15 zadań jest wymagane do zaliczenia kursu.

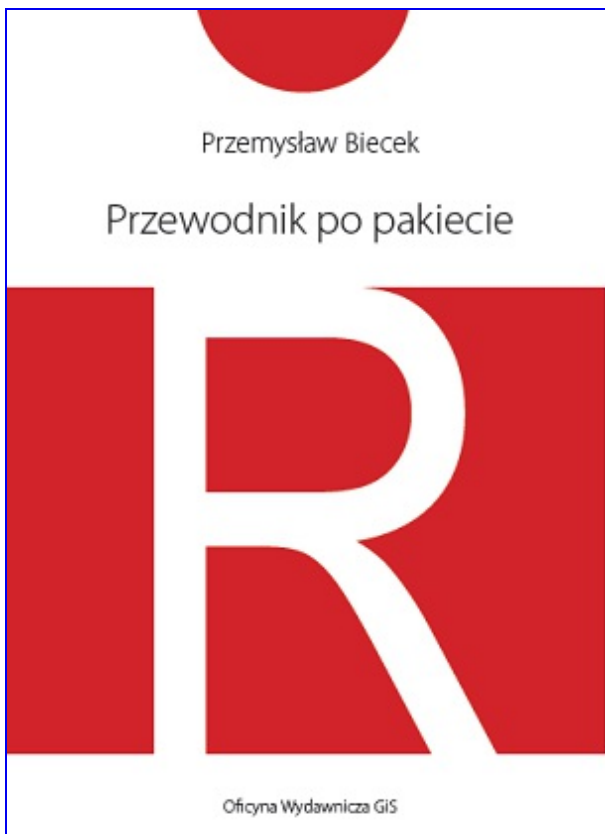
## Książki

### W języku polskim

Coraz więcej książek, podręczników i artykułów jest poświęconych programowi R.

W pierwszej kolejności chciałbym polecić pozycję, którą sam przygotowałem będąc doktorantem. Co jakiś czas ją uaktualniam, do dziś doczekała się ona trzeciego rozszerzonego wydania. Można z jej pomocą poznać podstawy języka R jak również bardziej zaawansowane mechanizmy takie jak pakiet *knitr* do automatycznego tworzenia raportów, funkcje do analizy danych, prezentowania danych, optymalizacji, przetwarzania wielkoskalowego oraz wiele innych tematów (omówionych jest ponad 500 funkcji dostępnych w R). Połowa tej książki dostępna jest bezpłatnie na poniższej stronie.

*Przewodnik po  
pakiecie R  
Przemysław  
Biecek*



<http://biecek.pl/R/>

Jedna z pierwszych pozycji poświęconych językowi R w języku polskim, do dziś dostępna bezpłatnie, to książka Łukasza Komsty dostępna pod adresem

*Wprowadzenie do środowiska R Łukasz Komsta*

<http://cran.r-project.org/doc/contrib/>

Dziś w języku polskim jest wiele książek, pisanych przez

autorów kładących nacisk na język, na zastosowania (czy to w biologii, ekonomii, czy innych dziedzinach) oraz na określone metody (analiza danych przestrzennych, szeregów czasowych, analiza regresyjna).

Pełna lista polskich książek jest rozwijana na stronie Wikipedii

[http://pl.wikipedia.org/wiki/R\\_\(j%C4%99zyk\\_program](http://pl.wikipedia.org/wiki/R_(j%C4%99zyk_program)

## **W języku angielskim**

W języku angielskim występuje zatrzesienie dobrych i słabych pozycji.

Dla osób zainteresowanych głębiej językiem szczególnie polecam otwartą książkę

*Advanced R* Hadley Wickham

<http://adv-r.had.co.nz/>

Lista ponad 150 innych książek o R znajduje się na stronie

<http://www.r-project.org/doc/bib/R-books.html>

W języku angielskim są również dostępne kursy online dotyczące R na takich serwisach jak

<https://www.coursera.org/> czy <https://www.edx.org/>



# Dla użytkowników Excela

Użytkownikom Execela, którzy chcieliby przejść płynnie z Excela do R polecam świetną książkę, dostępną w całości online opracowaną przez Johna Taverasa „R for Excel Users”.

Książce przyświeca miłe dla oka motto „Excel analysts who know R can do more with data”.

<http://www.rforexcelusers.com/book/preface/>

Oraz blog prowadzony przez Marco Ghislanzoni, na którym dostępnych jest wiele materiałów video porównujących jak pewne „Excelowe” rzeczy można zrobić w R.

<http://marcoghislanzoni.com/blog/>

## Blogi

### W języku polskim

W języku polskim o R można poczytać na stronach

<http://smarterpoland.pl/index.php/category/r/>

<http://thinking-in-r.blogspot.com/>

## **W języku angielskim**

Agregator wielu ciekawych blogów poświęconych R znajduje się na stronie

<http://www.r-bloggers.com/>

## **Spotkania użytkowników R**

### **W Polsce**

SER - Warszawa

<http://smarterpoland.pl/SER>

PAZUR - Poznań

<http://estymator.ue.poznan.pl/pazur/>

eRka - Kraków

<http://www.erkakrakow.pl/>

WZUR - Wrocław (już od jakiegoś czasu wstrzymany, ale kto wie)

<http://www.biecek.pl/WZUR/>

# Na świecie

Lista wielu grup użytkowników R na całym świecie:

<http://blog.revolutionanalytics.com/local-r-groups.html>