

Metody przetwarzania danych meteorologicznych w języku programowania R

Bartosz Czernecki

Spis treści

Wprowadzenie	7
1 R	9
1.1 Kilka słów o R ...	9
1.2 Dlaczego warto uczyć się R ?	11
1.3 X przykazań nauki R	12
1.4 Od czego zacząć?	13
2 Podstawy R	15
2.1 Terminal R	16
2.2 Arytmetyka	17
2.3 Tworzenie obiektów	18
2.4 Generowanie ciągów liczbowych	20
2.5 Łączenie obiektów	22
Zadania sprawdzające	23
3 Podstawy R - część 2	25
3.1 R skrypt	25
3.2 Praca ze skryptem	27
3.3 Wektory i podstawowe operacje na wektorach	28
3.4 Indeksowanie z użyciem warunków logicznych	33
3.5 Zadania podsumowujące	38
Zadanie domowe	40
Zadania sprawdzające	40
4 Ramki danych i macierze	41

4.1	Ramki danych	41
4.2	Praca na ramkach danych	42
4.3	Macierze	47
5	Wczyt i zapis danych	53
5.1	Katalog roboczy	54
5.2	Wczyt danych	55
5.3	Zapis danych	65
6	Pętle	69
6.1	Pętla <i>for</i>	70
6.2	Pętla <i>while</i>	75
7	Czas i napisy	79
7.1	Date	79
7.2	POSIXct	83
8	dplyr - część I	85
8.1	Łączenie ramek danych - <code>left_join()</code>	85
9	dplyr - część II	95
	Dane	95
9.1	Wybór kolumn - <code>select()</code>	97
9.2	Filtrowanie	99
9.3	Sortowanie - <code>arrange</code>	102
9.4	Przetwarzanie potokowe	104
9.5	<code>group_by()</code> oraz <code>summarise()</code>	108
	Zadanie sprawdzające	111
10	Postać wąska i szeroka	113
10.1	Postać wąska	113
10.2	Postać szeroka	116
10.3	Sklejanie i rozszczepianie kolumn	117
11	Grafika	119
11.1	graphics	119
11.2	ggplot2	121

<i>SPIS TREŚCI</i>	5
--------------------	---

Instalacja	125
Instalacja R	125
Instalacja RStudio	125
Instalacja bibliotek	127
Sprawy organizacyjne	129
Kolokwium końcowe - przykładowe zagadnienia:	129

Wprowadzenie

Przetwarzanie danych to niezwykle szeroka dyscyplina wymagająca praktycznego zastosowania szeregu umiejętności w celu poprawnego zrozumienia i interpretacji analizowanych zbiorów danych.

Każdego dnia gromadzone są petabajty nowych informacji związanych z monitoringiem atmosfery. Przetwarzanie choćby niewielkiego wycinka z nich wymaga często stosowania złożonych algorytmów postępowania. Te z kolei wymuszają stosowanie odpowiednich narzędzi obliczeniowych, których poznanie jest procesem długotrwałym, wykraczającym najczęściej poza pojedynczy podręcznik lub kurs.

Mając na uwadze powyższe przesłanki nadrzędnym celem niniejszego skryptu jest przede wszystkim zaznajomienie z “narzędziami” języka programowania R do analizy i wizualizacji danych, które pozwolą Tobie na dalsze samodoskonalenie nabytych umiejętności.

“Analiza danych, a w szczególności analiza z użyciem programu **R**, charakteryzuje się stromą krzywą uczenia. Na początku wiele rzeczy będzie nowych i trudnych. Gwarantuję jednak, że wysiłek włożony w poznawanie programu R opłaci się. Z czasem będziesz coraz sprawniej przetwarzać i wizualizować dane, a elastyczność i ekspresja programu R powodują, że praktycznie nie będzie przed Tobą barier związanych z analizą najróżniejszych danych” (Biecek, 2016) ... nie tylko meteorologicznych...

Rozdział 1

R

1.1 Kilka słów o R ...

R to bardzo dynamicznie rozwijający się język programowania używany przede wszystkim do analiz statystycznych, przetwarzania i wizualizacji danych. Jego rosnąca w ostatnich latach popularność jest związana przynajmniej z kilkoma czynnikami. Oto niektóre z nich:

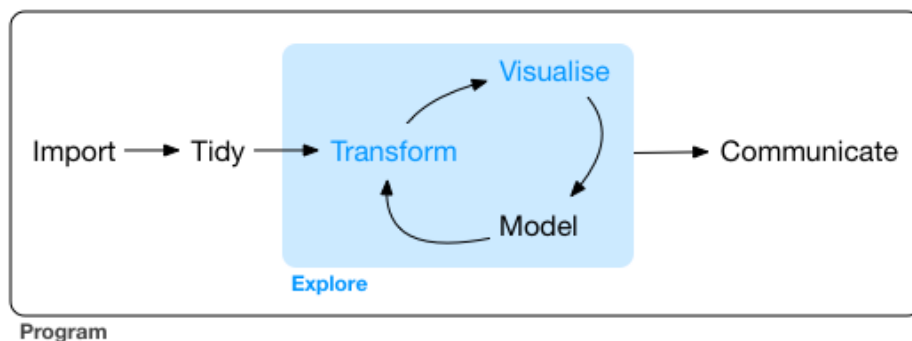
- W porównaniu do wielu innych języków programowania nauka **R** jest często dużo łatwiejsza (głównie dzięki relatywnie łatwej składni i interaktywności związanej z brakiem konieczności kompilacji).
- Popularność **R** mocno poszybowała w górę od czasu wydania zintegrowanego środowiska programistycznego (IDE) **RStudio**, które w znacznym stopniu usprawnia komfort pracy.
- Multiplatformowość - **R** można uruchamić na niemal każdym współczesnym systemie operacyjnym (Windows, Mac OS, Linux/UNIX), zarówno na komputerach PC, jak i na dużych klastrach obliczeniowych.
- **R** jest oprogramowaniem bezpłatnym i otwartoźródłowym - oznacza to, że korzystanie z **R** oraz modyfikowanie kodu na mocy licencji GPL nie nastręcza dodatkowych kosztów. Małe firmy tną w ten sposób koszty, giganci (m.in. Google, Yahoo, Facebook, Microsoft, NYSE, Wirtualna Polska, itp.)

dostosowują rozwiązania bazujące na **R** do własnych potrzeb.

- **R** jest językiem programowania, co oznacza że nawet jeśli nie ma interesującego nas algorytmu w natywnie zainstalowanym **R** możemy takie narzędzie stworzyć samemu. Najczęściej jednak okazuje się, że ktoś inny napisał już analogiczne lub podobne rozwiązanie i opublikował je w postaci pakietu programistycznego, który możemy bezpłatnie wykorzystać. Obecnie w serwisie CRAN znajduje się ponad 10000(!) takich pakietów dedykowanych dla szerokiego grona odbiorców.
- Liczba danych na których możemy jednocześnie pracować w standardowych rozwiązaniach ograniczona jest jedynie pamięcią RAM (dla wersji 64-bitowej). Obecnie jest to ok. 1-2 rzędy wielkości więcej w porównaniu do najczęściej stosowanych arkuszy kalkulacyjnych (Microsoft, 2016).
- W przypadku napotkania problemów możemy skorzystać z pomocy społeczności użytkowników **R**, którzy bardzo szybko reagują na forach dyskusyjnych (np. stackoverflow).
- ...

To tylko część z zalet związanych z stosowaniem środowiska programistycznego **R** do przetwarzania danych. Z punktu widzenia analizy danych meteorologicznych niezwykle istotne staje się wykorzystanie jednego środowiska pracy zamiast kilku oddzielnych aplikacji. W **R** możliwe jest jednoczesne zautomatyzowanie procesu pobierania i wstępnego przygotowania (czyszczenia) danych, rozbudowanej analizy statystycznej danych (także wielowymiarowych), wykorzystania różnych formatów danych (w tym np. GIS) i wykonania analiz czasowo-przestrzennych z finalną konwersją i wizualizacją danych w postaci statycznych lub interaktywnych rozwiązań.

W odróżnieniu od zwykle wykorzystywanych arkuszy kalkulacyjnych praca w **R** pozwala na dużą dozę automatyzacji całego procesu eksploracji danych od momentu ich importu do środowiska **R**, poprzez ich weryfikację i czyszczenie, transformację do wybranego formatu i skończywszy na etapie wizualizacji (ewentualnego modelowania/prognozowania) oraz końcowego przekazu do odbiorców końcowych (Wickham and Grolemund, 2016).



Rysunek 1.1: Typowy przebieg kolejnych etapów *data science* według Hadleya Wickhama (Wickham and Golemund, 2016)

1.2 Dlaczego warto uczyć się **R**?

Prawdopodobnie dotychczas do Twojej pracy z danymi wystarczała znajomość obsługi arkusza kalkulacyjnego, gdzie wszystkie dane swobodnie mieściły się w pojedynczym arkuszu i mogłeś je jednocześnie poukładać według własnego uznania i wizualizować za pomocą intuicyjnego, klikanego interfejsu graficznego. Filozofia pracy w **R** zdecydowanie różni się od powyższego schematu postępowania. **R** jest językiem programowania, co dla osób nie posiadających wcześniejszego przygotowania programistycznego oznacza mozolne poznawanie specyficznej składni i funkcji programistycznych.

Oznacza to także konieczność porzucenia własnych przyzwyczajeń. Jest to trudne, zwłaszcza na początku i wymaga wielu godzin wyłożonej pracy połączonej z twórczym eksperymentowaniem i korygowaniem niezliczonej liczby własnych błędów. Z pewnością jednak spędzony czas przy nauce **R** jest w relatywnie krótkim okresie rekompensowany z nawiązką, a analiza danych uprzednio zajmująca długie godziny często skraca się do czasu wykonywania pojedynczej linii kodu.

Umiejętność programowania w **R** jest coraz częściej doceniana na rynku pracy, co pokazuje rosnąca liczba ofert dla kandydatów z zaawansowaną obsługą środowiska **R**. Ten trend obserwuje się także w środowiskach naukowych oraz we wiodących ośrodkach badań atmosfery, gdzie **R** i blisko pokrewne języki wysokiego poziomu stają się *lingua franca* analizy danych.

1.3 X przykazań nauki R

Choć nie ma uniwersalnej recepty na naukę **R** warto pamiętać o poniższych wskazówkach, które pomogą w początkowych etapach pracy:

1. Nie bój się stromej krzywej uczenia (rys. 1.3). Efektywna nauka programowania wymaga długich godzin praktyki. Eksperymentuj z różnymi kombinacjami składni, które przyjdą Ci do głowy i sprawdzaj ich wyniki.
2. Interpretuj błędy pojawiające się po każdej błędnej komendzie.
3. Pracuj na własnych, dobrze znanych zbiorach danych. Łatwiej będzie Ci zrozumieć działanie poszczególnych funkcji i wyłapać ewentualne błędy.
4. Staraj się unikać początkowo dużych zbiorów danych jeśli nie jest to wymagane.
5. Korzystaj z systemu pomocy zarówno wbudowanej natywnie, jak i dostępnej online (google oraz stackoverflow).
6. Zrozumienie podstaw jest kluczowe aby analizować bardziej skomplikowane przypadki (dające dużo większą satysfakcję).
7. Twórz możliwie dużo i możliwie jak najbardziej opisowych komentarzy.
8. Staraj się utrzymywać odpowiedni porządek w składni tworzonego kodu oraz w nazwach plików.
9. R nie jest środowiskiem idealnym do wszystkich zastosowań. Prowadzenie budżetu domowego czy stworzenie pojedynczej, poprawnej kartograficznej mapy jest prawdopodobnie łatwiejsze i szybsze w innych programach.
10. Rób przerwy. Czasem najlepsze pomysły przychodzą w najmniej oczekiwanych momentach. Niekoniecznie przy komputerze.

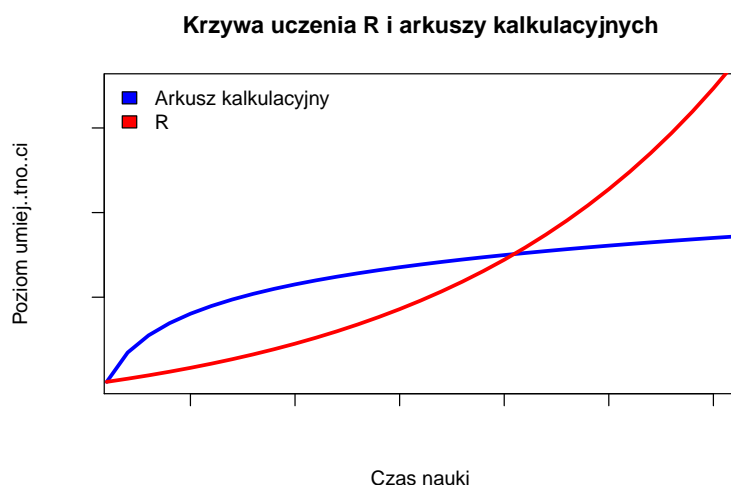
```
## Warning in title(...): conversion failure on 'Poziom umiejętności' in  
## 'mbcsToSbcs': dot substituted for <c4>
```

```
## Warning in title(...): conversion failure on 'Poziom umiejętności' in
```

```
## 'mbcsToSbcs': dot substituted for <99>

## Warning in title(...): conversion failure on 'Poziom umiejętności' in
## 'mbcsToSbcs': dot substituted for <c5>

## Warning in title(...): conversion failure on 'Poziom umiejętności' in
## 'mbcsToSbcs': dot substituted for <9b>
```



Rysunek 1.2: Porównanie stromej krzywej uczenia **R** i arkuszy kalkulacyjnych

1.4 Od czego zacząć?

Tematyce programowania w **R** poświęcono sporą liczbę podręczników, artykułów naukowych oraz *internetowych tutoriali* omawiających tajniki *data science*. Stanowią one cenne uzupełnienie niniejszego kursu, który wiele zagadnień (zwłaszcza technicznych) traktuje bardzo pobieżnie.

Spośród dostępnych źródeł książkowych opublikowanych w języku polskim na szczególną uwagę zasługują przede wszystkim podręczniki:

- Przewodnik po pakiecie R (Biecek, 2008) - najbardziej popularny podręcznik w Polsce, dostępny w kilku różnych wydaniach z których najłatwiejsze powinno być ostatnie (2017). Pierwsze rozdziały dostępne bezpłatnie na stronie autora biecek.pl/R.
- Programowanie w języku R. Analiza danych, obliczenia, symulacje (Gągolewski, 2016) - podręcznik zdecydowanie bardziej zaawansowany technicznie, rekomendowany dla osób mających wcześniejszy kontakt z programowaniem. Dostępny bezpłatnie ze strony internetowej biblioteki uniwersyteckiej UAM.
- Geostatystyka w R (Nowosad, 2016) - Książka opisująca rozszerzone standardy modelowania GIS. Dostępna bezpłatnie na stronie <https://bookdown.org/nowosad/Geostatystyka/>.
- Skrypt wprowadzający do R udostępniony na stronie internetowej Zakładu Klimatologii UAM - jest to bardzo krótkie wprowadzenie do **R** w dużym w zarysie prezentujące najważniejsze elementy niezbędne do pracy w tym środowisku. http://klimat.amu.edu.pl/?page_id=2500.

oraz

- An Introduction to R (Venables et al., 2004) - aktualizowany na bieżąco oficjalny podręcznik deweloperów (Team, 2016) omawiający podstawowe aspekty pracy w **R**.

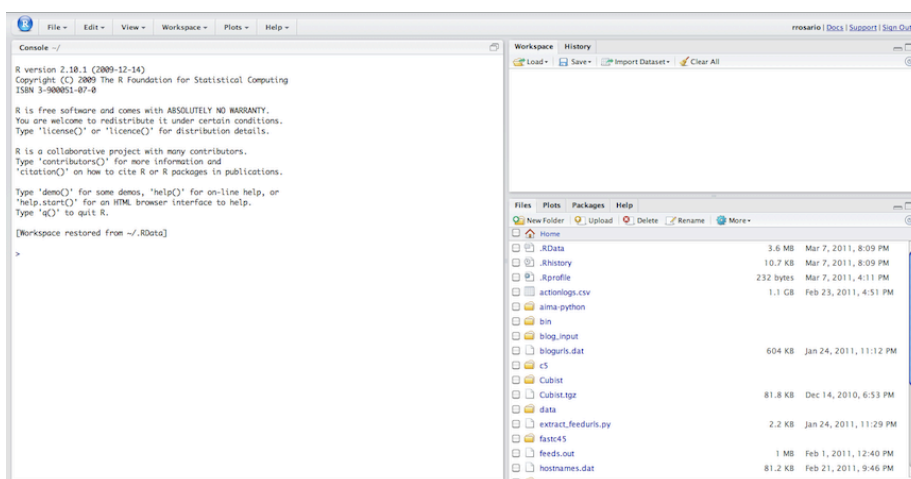
Kursy:

- Pogromcy danych: pogromcydanych.icm.edu.pl - dwuczęściowy kurs internetowy autorstwa Przemysława Biecka będący wprowadzeniem do zagadnień *data science* w **R**.
- Coursera: dostępnych przynajmniej kilka kursów internetowych związanych z przetwarzaniem danych z **R** [coursera.org](https://www.coursera.org).

Rozdział 2

Podstawy R

Praca w środowisku R możliwa jest w dwóch podstawowych trybach: skryptowym (wsadowym) oraz interaktywnym. Tryb interaktywny jest mniej skomplikowany i od niego rozpoczniemy naszą naukę. Praca w tym trybie polega na wprowadzaniu komend do konsoli (interpretera języka programowania), który znajduje się domyślnie po lewej stronie okna programu **RStudio**.



Rysunek 2.1: Ekran początkowy programu **RStudio**

2.1 Terminal R

W konsoli R możemy wpisywać komendy, a po naciśnięciu klawisza *Enter* komendy te są przez komputer interpretowane. Jeśli polecenie jest poprawne komputer obliczy jego rezultat. Alternatywnie informacja zwrotna o napotkanym błędzie wyświetlana jest jako *error* lub w formie ostrzeżenia (ang. *warning*). Spróbujmy wykorzystać R jako kalkulator i przetestujmy zachowanie terminala.

```
5+3
```

```
## [1] 8
```

Brak informacji o błędzie oznacza poprawne wykonanie działania. Obok wyniku w nawiasie kwadratowym komputer zwrócił liczbę porządkową pierwszej wartości w danym wierszu.

```
5+2,5
```

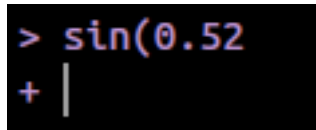
```
## Error: <text>:1:4: unexpected ' ','  
## 1: 5+2,  
##      ^
```

Komunikat błędu wskazuje miejsce jego wystąpienia. W tym przypadku jest to przecinek, który nie pasuje do składni interpretowanego polecenia, ponieważ separatorem miejsc dziesiętnych w R jest kropka a nie przecinek.

Jeśli chcemy szybko poprawić ten błąd możemy użyć kursorów (strzałek) na klawiaturze góra-dół do przeglądania ostatnio wprowadzonych poleceń i kursorami lewo-prawo przenieść się do miejsca wymagającego poprawy.

Zwróć uwagę, że jeśli komenda jest (przynajmniej częściowo) poprawna, ale nie zostanie zakończona w danej linii, wówczas po naciśnięciu *Enter*a zamiast tzw. znaku zachęty ">" zostanie zwrócony znak "+".

W takiej sytuacji możliwe są 2 rozwiązania - dokończenie wpisywania poprzedniej komendy, lub naciśnięcie klawisza *Esc* w celu przerwania bieżącego procesu. Klawisz *Esc* (lub ikonkę symbolizującą znak *STOP*) można nacisnąć zawsze w celu przerwania aktualnie aktywnego procesu.



Rysunek 2.2: Przykład niepoprawnie zakończonej komendy (brak znaku “)”), i pojawienie się znaku “+” w nowej linii konsoli oznaczający możliwość dokończenia wpisywanej komendy

Zadanie:

Sprawdź powyższe działania terminala testując polecenie: `6+` (*Enter*), w kolejnej linii wprowadź dowolną liczbę i ponownie naciśnij *Enter*. Za drugim razem wpisz `6+` (*Enter*), ale tym razem przerwij działanie stosując *Esc*.

2.2 Arytmetyka

Wykonaj poniższe zadania sprawdzające działanie **R** jako kalkulatora:

1. Oblicz wyrażenie `2+3*5`
2. Stosując nawiasy, które pozwalają na zmianę domyślnej kolejności wykonywania działań zmodyfikuj powyższe działanie w taki sposób, aby najpierw była wykonywane sumowanie, a dopiero potem mnożenie (tj., końcowy wynik powinien dać 25)
3. Pomnóż liczby 3 i 5
4. Podziel dowolne 2 liczby
5. Sprawdź działanie operatorów `%` oraz `%%` na dowolnych dwóch liczbach całkowitych. Do czego one służą?
6. Sprawdź działanie operatora `^`

Jeśli chcesz obliczyć wynik funkcji wykładniczej (`exp`), sinusa (`sin`), cosinusa (`cos`), pierwiastka kwadratowego (`sqr`), logarytmu dziesiętnego (`log`), wartości bezwzględnej (`abs`), wartości maksymalnej (`max`), minimalnej (`min`), średniej (`mean`), odchylenia standardowego (`sd`), itp., musisz zastosować składnię funkcyj-

na, o której powiemy nieco później. W najprostszej postaci należy wprowadzić nazwę funkcji i w nawiasie podać argument dla funkcji, czyli w tym przypadku liczbę.

UWAGA! Wielkość znaków w **R** MA ZNACZENIE! (czyli `LOG` to dla komputera coś innego niż `log`).

W RStudio bardzo przydatny jest klawisz tabulatora, który pozwala na uzupełnienie nazwy funkcji lub obiektu po naciśnięciu klawisza *TAB*. W ten sposób np. po wpisaniu liter `sq` i naciśnięciu *TAB* pojawi się intuicyjne okno z podpowiedzią.

7. Na podstawie powyższych informacji oblicz:

- Pierwiastek z 9
- Cosinus liczby `pi`, gdzie `pi` jest stałą wbudowaną w R (wystarczy wpisać `pi` zamiast 3.141593)

8. W jednej linii kodu podnieś liczbę 9 do kwadratu i oblicz pierwiastek kwadratowy (`sqr`) z tej liczby. Podpowiedź: funkcje mogą być wzajemnie zagnieżdżone

Wskazówka: Przy interpretacji polecenia R pomija spacje pomiędzy poszczególnymi elementami działania. NIE JEST istotne czy twoja komenda jest zapisana jako `5+4.5` czy `5 + 4.5` (ale nie możesz wpisać spacji przy liczbach, np.: `5+4 .5`!).

2.3 Tworzenie obiektów

Dotychczas wynik naszego polecenia wyświetlał się na ekranie, ale nie mogliśmy z nim dalej nic zrobić, bo nie był zapisywany w pamięci komputera. Wpisywanie komend w terminalu i sprawdzanie ich wyniku nie jest zbyt efektywne, dlatego też większość pracy w **R** odbywa się na obiektach. Czym jest obiekt? W bardzo dużym skrócie to po prostu nazwa do której przypisuje się wartość. Nazwa nie może się zaczynać od liczb oraz znaków specjalnych. Najlepiej także unikać polskich znaków.

W **R** istnieje kilka sposobów tworzenia obiektów (zmiennych). Najczęściej stosowanym operatorem przypisania jest wyrażenie `<-`, które w **RStudio** możemy

otrzymać za pomocą skrótu lewy **Alt** + **-**. Warto dobrze zapamiętać ten skrót.

Obiekt można stworzyć także za pomocą operatora `=`, który działa analogicznie jak `<-`. Ze względu na pewne uwarunkowania historyczno-techniczne bardziej rekomendowany jest zapis w postaci strzałki. Wyrażenie można przypisać także prawostronnie za pomocą operatora `->`. Choć jest to poprawna forma zapisu, w praktyce jest rzadko stosowana.

Stwórzmy zatem naszą pierwszą zmienną, którą nazwiemy **temperatura** i przypiszmy jej wartość 279.15:

```
temperatura <- 279.15
```

Zauważ, że w prawym górnym rogu okna **RStudio** w zakładce **Environment** pojawiła nazwa zdefiniowanej zmiennej i jej wartość. W tej zakładce będą pojawiać się wszystkie nazwy stworzonych lub wczytanych obiektów.

W odróżnieniu od wcześniejszych zastosowań **R** jako kalkulatora nie wyświetlił nam się wynik tej operacji. To dlatego, że zapisaliśmy go w pamięci komputera. Brak informacji zwrotnej jednocześnie oznacza, że operacja przebiegła poprawnie.

Jeśli chcemy wyświetlić zawartość naszej zmiennej możemy wpisać po prostu jej nazwę lub wykorzystać funkcję `print` (pamiętaj o używaniu *TABulatora*):

```
temperatura
```

```
## [1] 279.15
```

```
print(temperatura)
```

```
## [1] 279.15
```

Pamiętaj, że wielkość znaków w **R** MA ZNACZENIE, tzn. wyrażenie **temperatura** i **Temperatura** to dla komputera 2 różne obiekty!

Wyobraźmy sobie, że wartość przechowywana w obiekcie **temperatura** to temperatura powietrza wyrażona w Kelwinach (podstawowa jednostka układu SI). Jeśli chcemy przeliczyć Kelwiny na stopnie Celsjusza musimy zastosować poniższe równanie (2.1):

$$T(^{\circ}C) = K - 273.15 \quad (2.1)$$

W tym momencie można wykorzystać zawartość zmiennej `temperatura` aby podstawić ją do równania i zapisać wynik działania do obiektu, który nazwiemy `tc` (temperatura w Celsjuszach):

```
tc <- temperatura-273.15
```

Zadanie:

1. Stosując poniższy wzór na przeliczenie temperatury ze stopni Celsjusza na temperaturę w Fahrenheitach utwórz obiekt `tf`. Wartość temperatury przelicz ze zmiennej `tc`

$$T(^{\circ}F) = T(^{\circ}C) * 1.8 + 32 \quad (2.2)$$

2. Na nowym obiekcie `tf` sprawdź działania funkcji matematycznych `floor`, `ceiling` oraz `round`.
3. Sprawdź co stanie się po wywołaniu komendy `?round` oraz `??"ceiling"`?

2.4 Generowanie ciągów liczbowych

Praca na obiektach przechowujących jedną wartość nie daje zbyt wielkich korzyści. Obiekty R mogą przechowywać znacznie więcej wartości i aby się o tym przekonać poznamy 3 podstawowe schematy generowania ciągów liczbowych:

1. Za pomocą `:` możemy stworzyć ciąg liczb z interwałem co 1. W zależności od tego jakie wartości podstawimy z prawej i lewej strony dwukropka będzie to ciąg rosnący lub malejący:

```
150:180
```

```
## [1] 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166
```

```
## [18] 167 168 169 170 171 172 173 174 175 176 177 178 179 180
```

```
5:-20
```

```
## [1] 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11
```

```
## [18] -12 -13 -14 -15 -16 -17 -18 -19 -20
```

```
11.375:34
```

```
## [1] 11.375 12.375 13.375 14.375 15.375 16.375 17.375 18.375 19.375 20.375
```

```
## [11] 21.375 22.375 23.375 24.375 25.375 26.375 27.375 28.375 29.375 30.375
```

```
## [21] 31.375 32.375 33.375
```

2. Podobnie do powyższego schematu działa funkcja `seq`. Szczegóły jej działania można znaleźć po wywołaniu komendy `?seq` (dostęp do systemu pomocy). Zgodnie z uzyskanymi informacjami z systemu pomocy - jeśli chcemy wygenerować ciąg liczb od 0 do 20 co 1 komenda będzie wyglądała następująco

```
seq(from=0, to=20, by=1)
```

```
## [1] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Jeśli nie wiemy jaki powinien być interwał pomiędzy liczbami, ale znamy długość generowanego ciągu opcję `by` możemy zastąpić opcją `length.out`. Jeśli chcemy wygenerować 35 liczb w równych odległościach od 0 do 20, wówczas możemy zastosować poniższy kod:

```
seq(from=0, to=20, length.out =35)
```

```
## [1] 0.0000000 0.5882353 1.1764706 1.7647059 2.3529412 2.9411765
```

```
## [7] 3.5294118 4.1176471 4.7058824 5.2941176 5.8823529 6.4705882
```

```
## [13] 7.0588235 7.6470588 8.2352941 8.8235294 9.4117647 10.0000000
```

```
## [19] 10.5882353 11.1764706 11.7647059 12.3529412 12.9411765 13.5294118
```

```
## [25] 14.1176471 14.7058824 15.2941176 15.8823529 16.4705882 17.0588235
```

```
## [31] 17.6470588 18.2352941 18.8235294 19.4117647 20.0000000
```

3. Losowe ciągi liczbowe można także wygenerować za pomocą funkcji z wybranych rozkładów statystycznych. Jednym z najbardziej przydatnych jest rozkład jednostajny ciągły, który w **R** jest wbudowany do funkcji `runif` lub rozkład normalny (`rnorm`). Przykładowo, jeśli chcemy wygenerować 5

losowych liczb w przedziale od 1 do 10, wówczas kod wygląda następująco:

```
runif(5, min=1, max=10)
```

```
## [1] 3.147456 1.836970 6.671314 6.033624 2.028277
```

Zadanie:

1. W arkuszach kalkulacyjnych można wygenerować narastający ciąg liczbowy chwytając za róg komórki z wartością i przesuwając ją w dół. Gdybyś chciał wygenerować liczby od 1 do samego dołu arkusza kalkulacyjnego zajęło by to prawdopodobnie dużo czasu. Stwórz analogiczny schemat postępowania w **R** aby wygenerować ciąg liczb od 1 do 1048576 i zapisz ten wynik do obiektu **a**. Które z rozwiązań jest szybsze?
2. *Dla chętnych:* Sprawdź ile danych w ciągu narastającym od 1 możesz wygenerować i zapisać do obiektu **a** zanim braknie komputerowi pamięci? Do testów wykorzystaj wielokrotność potęgi 2 (np. 2^{20} , 2^{35}). Ile było by potrzebnych arkuszy kalkulacyjnych do przechowania takiej liczby danych gdyby każda liczba była zapisana w nowym rzędzie?
3. Stworzony obiekt niemal w całości wypełnia dostępną pamięć RAM naszego komputera. Usuń stworzony obiekt za pomocą funkcji `rm(a)`
4. Korzystając ze wzoru przeliczającego temperaturę ze stopni Celsjusza na stopnie Fahrenheita (2.2) wygeneruj wartości temperatury w stopniach Celsjusza od -10 do +30 i oblicz ich wartości w stopniach Fahrenheita
5. Wygeneruj losowo 50 liczb o wartościach od 0 do 1 z rozkładu jednostajnego i oblicz wartość średnią. Funkcja do obliczania średniej arytmetycznej nazywa się `mean`.

2.5 Łączenie obiektów

Tworzenie obiektów jest możliwe także za pomocą operatora `c()`, który w **R** pozwala na sklejanie dwóch lub więcej wyrażeń (np. liczbowych) w jedno. Fachowo

taka operacja złączająca nazywa się *konkatenacją*.

Funkcja `c()` jest bardzo prosta w swoim działaniu i wymaga podania kolejnych złączanych obiektów rozdzielonych przecinkami. Najlepiej zaznajomić się z *konkatenacją* na przykładach:

1. Utwórz obiekt `a`, który będzie przechowywał wartości liczbowe 1,2,3,5

```
a <- c(1,2,3,5)
```

2. Utwórz obiekt `b`, który będzie zawierać wartości od 0 do 10 co 1 oraz od 10 do 1 co 1. Wykorzystaj do tego celu operator :

```
b <- c(0:10, 10:1)
```

3. Złącz obiekty `a` i `b`, na końcu dołącz obiekt `b` powiększony o 20

```
c(a, b, b+20)
```

```
## [1] 1 2 3 5 0 1 2 3 4 5 6 7 8 9 10 10 9 8 7 6 5 4 3  
## [24] 2 1 20 21 22 23 24 25 26 27 28 29 30 30 29 28 27 26 25 24 23 22 21
```

4. Oblicz odchylenie standardowe (z ang. *standard deviation*) liczb 0,5,-2,4

```
sd(c(0,5,-2,4))
```

```
## [1] 3.304038
```

Zwróć uwagę, że aby wykonać prawidłowo operację na obiekcie zawierającym więcej niż 1 element konieczna jest jego wcześniejsza konkatenacja (jak w powyższym przykładzie).

Zadania sprawdzające

<https://goo.gl/forms/TXfljFnFHwSXHgji1>

Rozdział 3

Podstawy R - część 2

3.1 R skrypt

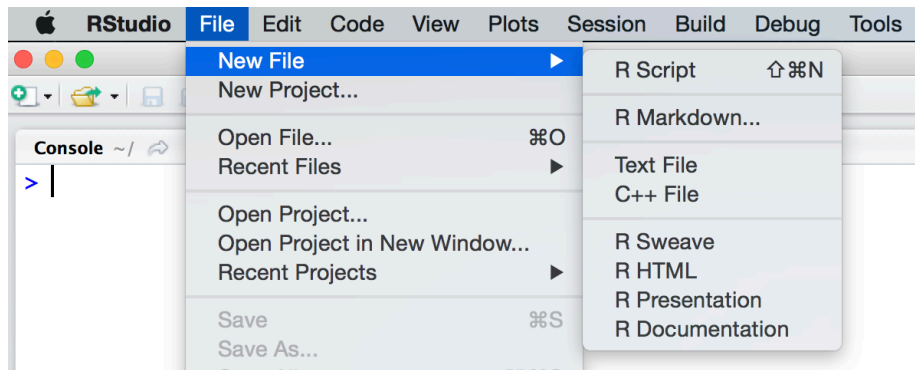
Na poprzednich zajęciach praca z R opierała się na wprowadzaniu komend do terminala, gdzie były one natychmiast przetwarzane przez komputer. W praktyce rzadko się zdarza, że cały proces przetwarzania danych czy tworzenia własnego algorytmu można sprowadzić do kilku- kilkunastu linii kodu. Dlatego też ciąg wydawanych poleceń zapisywany jest najczęściej jako plik skryptowy R, który pozwala na:

- łatwiejsze uporządkowanie kolejnych poleceń w logiczny ciąg przetwarzania danych
- w razie znalezienia błędu - łatwiej jest poprawić taki kod widząc cały schemat postępowania od początku do końca
- każdą linię kodu możemy automatycznie uruchomić i sprawdzić jaki daje wynik.

Utworzenie nowego pliku skryptowego w programie **RStudio** możliwe jest za pomocą:

- wybrania z górnego menu opcji **File**, następnie opcji **New File** oraz **R Script** (jak pokazano w przykładzie)
- poprzez skrót klawiszowy **ctrl+shift+n**

- wybierając ikonę białej kartki z zielonym kółkiem i znakiem plusa. Ikona znajduje się skrajnie z lewej w pierwszym rzędzie ikon. Po wysunięciu szeregu opcji należy wybrać pierwszą z nich, tj. **R Script**.



Rysunek 3.1: Tworzenie nowego skryptu R w interfejsie programu **RStudio**

Lewe okno programu **RStudio** powinno podzielić się na 2 mniejsze, z dotychczasowym terminalem na dole i nowym oknem do wpisywania skryptu u góry.

Plik skryptowy R jest zwykłym plikiem tekstowym z rozszerzeniem `.r` lub `.R`. Określenie “zwykły plik tekstowy” oznacza, że zawiera on tylko te informacje, które są widoczne na ekranie. To trochę tak, jakby wszystkie nasze dotychczasowe komendy wpisywane w terminalu zapisać w “notatniku”, a następnie wklejać je w zależności od potrzeb do terminala.

Jak działa skrypt R?

Aby “przenieść” daną linię kodu z okna skryptowego do terminala i ją wykonać należy posłużyć się skrótem `ctrl+enter`. Wówczas aktywna linia kodu zostanie przeniesiona do terminala i wykonana.

Jeśli chcemy jednorazowo uruchomić więcej niż 1 linię kodu (np. jedno polecenie rozpisałyśmy w kilku liniach) możemy zaznaczyć je myszką (lub za pomocą strzałek i `shift`), a następnie wykonać ten blok kodu stosując skrót `ctrl+enter`. Częste stosowanie tego skrótu sprawi, że część osób nazwie te zajęcia “ctrl+enter” ;)

Docelowo będziemy dążyć do postaci, w której cały stworzony skrypt stanowi

spójną całość i może być uruchomiony linia po linii od początku do końca. Zamiast wielokrotnego wciskania `ctrl+enter` linia po linii (lub zaznaczenia całego kodu i naciśnięcia `ctrl+enter`) można ten sam efekt uzyskać za pomocą ikony **Source** znajdującej się w prawym górnym rogu okna skryptowego (skrót: `ctrl+shift+s`). Jest to równoznaczne z użyciem funkcji `source(nazwa_naszego_pliku_skryptowego.R)`.

3.2 Praca ze skryptem

Praca ze skryptami wymaga wyrobienia pewnych nawyków, które pozwolą na bardziej efektywne tworzenie kodu i przetwarzanie danych.

3.2.1 Komentarze

Komentarz to fragment kodu znajdujący się bezpośrednio po znaku `#`, który nie jest interpretowany przez komputer podczas wykonywania danego polecenia. W środowisku **RStudio** najczęściej po znaku komentarza zmienia się kolor składni dający do zrozumienia użytkownikowi, że dalszy fragment kodu to właśnie komentarzem:

```
length(-5:5) # funkcja "length" zwraca liczbę elementów / tu: długość ciągu liczbowego
```

```
## [1] 11
```

Dobrym zwyczajem, zwłaszcza na początku nauki programowania i/lub przetwarzania danych jest tworzenie możliwie obszernych i precyzyjnych komentarzy ułatwiających zrozumienie naszego działania lub poszczególnych bloków kodu.

Jedna ze szkół programowania zakłada, że “z komentarzy powinno się móc wnioskować wszystko co program robi, bez oglądania reszty źródeł” (Dewhurst and Stark, 1995). Stosowanie tej zasady pozwoli na dużą oszczędność czasu, zwłaszcza jeśli nasz skrypt:

- jest używany do nauki programowania/przetwarzania danych,
- jest stosunkowo skomplikowany,
- zawiera wiele nowych elementów,
- jest otwierany stosunkowo rzadko,

- ma być docelowo używany także przez inne osoby.

Komentarze warto stosować także w odniesieniu do pewnych fragmentów testowanego kodu, które chcemy chwilowo wyłączyć, ale nie chcemy się ich na stałe pozbywać. Twórcy **RStudio** pomyśleli o takim zastosowaniu dając użytkownikom skrót `ctrl+shift+c`, który w zaznaczonym fragmencie kodu na początku każdej linii tworzy znak komentarza. W celu przetestowania tego rozwiązania stwórz poniższy fragment kodu nawiązujący do przeliczeń temperatury powietrza z stopni Celsjusza na stopnie Fahrenheita:

```
tc <- -20 # wartosc temperatury w *C
tf <- tc*1.8+32 # przeliczamy na stopnie F
print(tf) # wyświetlamy zawartość zmiennej tf

tc <- -10:10 # za drugim razem chcemy przetestować zakres wartości temperatury w *C od
tf <- tc*1.8+32 # znowu przeliczamy na stopnie F
print(tf) # wyświetlamy zawartość zmiennej tf
```

A następnie zaznacz ostatnie 3 linie kodu i zakomentuj skrótem `ctrl+shift+c`. Uruchom cały kod (łącznie z zakomentowanymi liniami) i dla pewności sprawdź wartości przechowywane w zmiennych.

3.3 Wektory i podstawowe operacje na wektorach

Podstawowym typem obiektu w **R** są wektory. Znasz je już z poprzednich ćwiczeń, kiedy traktowaliśmy **R** jako kalkulator, gdy definiowaliśmy nowe obiekty lub generowaliśmy ciągi liczb. Nawet pojedyncza wartość w **R** jest w rzeczywistości wektorem (1-elementowym), natomiast jeśli chcemy dowolne operacje wykonywać na większej liczbie elementów wówczas musimy te obiekty złączyć za pomocą funkcji `c()`.

Warto zaznajomić się z podstawowymi cechami pracy na wektorach w celu uniknięcia późniejszych błędów.

Zadanie

1. Stwórz wektor `x` o wartościach 0, 3, 2, 10, 5 oraz wektor `y` o wartościach 3, 4, 5. Wykonując operację dodawania, mnożenia i potęgowania na tych dwóch obiektach sprawdź jak działa *autoreplikacja* w **R**.
2. Zmodyfikuj obiekt `y` tak aby zawierał tyle elementów co zmienna `x`, ale jako 4-ty i 5-ty element wprowadź wartości `NA` oznaczające brak danych. Ponownie przetestuj mnożenie na tych dwóch obiektach i sprawdź działanie autoreplikacji.
3. Przetestuj działanie funkcji `mean()` oraz `sum()` i `cumsum()` na obiekcie `y`. Korzystając z pomocy systemowej sprawdź jak “naprawić” wynik, tak aby uwzględniał on przy obliczaniu tylko wartości liczbowe.
4. Sprawdź działanie funkcji `sort()` na zmiennej `x`.
5. Sprawdź działanie funkcji `rev()` na zmiennej `y`.

Staraj się przyswoić możliwie wiele funkcji omówionych do tej pory. W praktyce najczęściej stosuje się kilkadziesiąt słów kluczowych, które pozwalają na rozwiązanie większości spotykanych problemów obliczeniowych.

3.3.1 Typy danych wektorowych

W **R** wektory mogą przechowywać nie tylko liczby, ale także typy danych czynnikowych, datę, czas oraz ciągi tekstowe i logiczne. Póki co omówimy w dużym skrócie te 2 ostatnie, które przydadzą się w kolejnym podpunkcie.

3.3.1.1 Typ tekstowy

Deklaracja typu tekstowego odbywa się poprzez wpisanie dowolnego wyrażenia między znaki cudzysłowia " " lub w ciapkach ' '. Jeśli chcemy stworzyć obiekt przechowujący pierwsze litery alfabetu możemy go zdefiniować w następujący sposób:

```
c("a", "b", "c")
```

```
## [1] "a" "b" "c"
```

Warto zwrócić uwagę, że jeśli będziemy próbowali złączyć obiekty o różnych typach **R** domyślnie postara się je “sprowadzić” do postaci najbardziej ogólnej,

co często wymusza przekonwertowanie jednego typu danych w inny:

```
c(1:5, "0", "5")
```

```
## [1] "1" "2" "3" "4" "5" "0" "5"
```

W powyższym przykładzie można rozpoznać konwersję liczb do typu tekstowego ponieważ wszystkie elementy **R** wydrukował w cudzysłowach.

Wniosek: w wektorze możemy przechowywać TYLKO jeden typ danych!

3.3.1.2 Typ logiczny

Wiele procedur przetwarzania danych wymaga sprawdzania warunków logicznych, które mogą przyjmować wartości PRAWDA lub FAŁSZ. W środowisku **R** te wartości są deklarowane za pomocą słów TRUE lub FALSE, które można sprowadzić do skróconego zapisu dużymi literami T i F.

```
c(TRUE, FALSE, TRUE, FALSE) # pełne słowa
```

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

```
c(T, F, T, F) # to samo, ale w skróconym zapisie
```

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

Ciekawą własnością typu logicznego jest to, że po sprowadzeniu do wartości numerycznej (np. funkcją `as.numeric()`) przyjmuje ona wartości 1 (prawda) lub 0 (fałsz):

```
logiczny <- c(TRUE, FALSE)
as.numeric(logiczny)
```

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

3.3.2 Indeksowanie wektorów

Indeksowanie wektorów polega na wyborze określonych elementów obiektu poprzez wskazanie ich pozycji. Indeksy wektora w **R** numerowane są od 1 do n,

gdzie n to długość (liczba elementów) wektora. Jeśli nie znasz długości wektora zawsze możesz ją sprawdzić za pomocą funkcji `length()`.

Wybór dowolnych elementów wektora jest możliwy przez zastosowanie operatora `[]` wpisując jako argument pozycję, którą chcemy pobrać. Na początek stwórzmy wektor zawierający 20 losowych liczb z przedziału od 0 do 1. Nazwijmy ten obiekt `dane` i wyświetlmy jego zawartość:

```
dane <- runif(20)
dane

## [1] 0.899049622 0.784734013 0.595490900 0.300648547 0.234256251
## [6] 0.302369527 0.337933923 0.780297597 0.012236627 0.389173037
## [11] 0.762023591 0.549608661 0.857791153 0.259316680 0.857080017
## [16] 0.255409679 0.247717798 0.326963206 0.007979414 0.259561501
```

Komenda pozwalająca na pobranie np. tylko piątego elementu:

```
dane[5]

## [1] 0.2342563
```

Do wyświetlenia np. tylko pierwszych 10-ciu elementów musimy w nawiasie kwadratowym wprowadzić numery indeksów od 1 do 10. Wymaga to wygenerowania ciągu liczb, który będzie traktowany przez **R** jako ciąg liczb 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Można ten efekt osiągnąć przynajmniej na 2 sposoby.

1. Poprzez konkatencję `c()`:

```
dane[ c(1,2,3,4,5,6,7,8,9,10) ]

## [1] 0.89904962 0.78473401 0.59549090 0.30064855 0.23425625 0.30236953
## [7] 0.33793392 0.78029760 0.01223663 0.38917304
```

2. Poprzez stworzenie ciągu liczb za pomocą operatora `:`

```
dane[ 1:10 ]

## [1] 0.89904962 0.78473401 0.59549090 0.30064855 0.23425625 0.30236953
## [7] 0.33793392 0.78029760 0.01223663 0.38917304
```

W analogiczny sposób możemy także odwrócić kolejność elementów obiektu `dane` wyświetlając zbiór wartości malejących od 20 do 1 z interwałem co 1:

```
dane[20:1]
```

```
## [1] 0.259561501 0.007979414 0.326963206 0.247717798 0.255409679
## [6] 0.857080017 0.259316680 0.857791153 0.549608661 0.762023591
## [11] 0.389173037 0.012236627 0.780297597 0.337933923 0.302369527
## [16] 0.234256251 0.300648547 0.595490900 0.784734013 0.899049622
```

Możemy także spróbować wyświetlić tylko wskazane przez nas indeksy. Jeśli chcemy wyświetlić 2 razy 10-ty element, następnie 2-gi, a potem 19-ty to możemy ponownie wykorzystać funkcję `c()`, która pozwala na stworzenie wektora składającego się z dowolnej konfiguracji indeksów.

```
dane[ c(10,10,2,19) ]
```

```
## [1] 0.389173037 0.389173037 0.784734013 0.007979414
```

W indeksowaniu ważne jest aby wyrażenie znajdujące się w `[]` było wektorem! Jeśli nie nabrałeś jeszcze wprawy w tworzeniu wektorów za pomocą `c()` oraz `:` zawsze możesz przetestować roboczo działanie wprowadzanego wyrażenia indeksującego w konsoli

W praktyce często do indeksowania wykorzystuje się inne obiekty (wektory), które zawierają numery pozycji do pobrania. Sprawdźmy to na przykładzie:

```
indeks <- c(1,5,10) # tworzymy wektor wartości 1, 5, 10
```

```
dane[indeks] # wyświetlamy wskazane numery, które są przechowywane w obiekcie `indeks`
```

```
## [1] 0.8990496 0.2342563 0.3891730
```

3.3.2.1 Indeksowanie przez “negację”

Ciekawym rozwiązaniem pozwalającym w wielu przypadkach na pozbycie się niechcianych elementów jest zastosowanie jako indeksu liczb ujemnych. Jeśli chcemy pozbyć się tylko np. pierwszego elementu z naszego obiektu (np. zawiera błąd), wówczas zamiast wpisywać komendę `dane[2:19]` możemy ten sam efekt osiągnąć stosując poniższy zapis:

```
dane[-1]
```

```
## [1] 0.784734013 0.595490900 0.300648547 0.234256251 0.302369527
```



```
## [6] 0.337933923 0.780297597 0.012236627 0.389173037 0.762023591
## [11] 0.549608661 0.857791153 0.259316680 0.857080017 0.255409679
## [16] 0.247717798 0.326963206 0.007979414 0.259561501
```

Wybór poprzez eliminację nie musi się ograniczać do pojedynczego indeksu. Możliwe jest także stworzenie wektora dowolnych liczb ujemnych:

```
dane[c(-1,-20)] # pozbywamy sie pierwszego i ostatniego elementu
```

```
## [1] 0.784734013 0.595490900 0.300648547 0.234256251 0.302369527
## [6] 0.337933923 0.780297597 0.012236627 0.389173037 0.762023591
## [11] 0.549608661 0.857791153 0.259316680 0.857080017 0.255409679
## [16] 0.247717798 0.326963206 0.007979414
```

```
dane[c(-5:-10, -15:-20)] # pozbywamy sie elementow od 5. do 10. oraz od 15. do 20.
```

```
## [1] 0.8990496 0.7847340 0.5954909 0.3006485 0.7620236 0.5496087 0.8577912
## [8] 0.2593167
```

3.3.2.2 Indeksowanie za pomocą wyrażeń logicznych

Ciekawym rozwiązaniem przy indeksowaniu może być zastosowanie wartości logicznych TRUE lub FALSE. Na początek stwórzmy nowy wektor zawierający nazwy 4-ech losowych miast i nazwijmy go `miasta`.

```
miasta <- c("Poznań", "Wrocław", "Warszawa", "Kraków")
```

Stosując wektor logiczny o wartościach TRUE lub FALSE możemy indeksować, które elementy mają zostać pobrane:

```
miasta[c(TRUE, TRUE, FALSE, TRUE)]
```

```
## [1] "Poznań" "Wrocław" "Kraków"
```

3.4 Indeksowanie z użyciem warunków logicznych

Indeksowanie jest niezmiernie ważnym elementem pracy z danymi w **R**. Szybkie i efektywne przetwarzanie danych wymaga testowania warunków logicznych

(o tym szerzej w kolejnych częściach kursu), które pozwalają na odfiltrowanie zbiorów danych na których chcemy wykonać naszą analizę. Najczęściej sprowadza się to do stworzenia nowych obiektów (wektorów) przechowujących numery pozycji lub wartości logiczne spełniające dany test. Poniżej zamieszczono tabelę z podstawowymi operacjami logicznymi w **R**:

Tabela 3.1: Podstawowe operatory logiczne w **R**

Operator	Działanie operatora
<	mniejsze od
<=	mniejsze bądź równe
>	większe od
>=	większe bądź równe
==	równe
!=	różne od
!x	negacja
x y	suma logiczna zbiorów
x & y	iloczyn logiczny zbiorów

3.4.1 Testowanie warunków logicznych

Działanie operatorów logicznych najlepiej sprawdzić w praktyce. Na początek stwórzmy obiekt `dane2`, w którym będziemy przechowywać 30 losowych wartości od -1 do +1 i wyświetlimy jego zawartość

```
dane2 <- runif(n = 30, min = -1, max = 1)
dane2
```

```
## [1]  0.18754995  0.37016365  0.39507023  0.91508907 -0.53614377
## [6]  0.25714506  0.82342568  0.55519429 -0.12100064 -0.23853956
## [11] -0.68655516  0.98477907  0.17569978 -0.61603732  0.79882611
## [16] -0.54385309  0.88125978 -0.70620560 -0.39978640 -0.12945671
## [21] -0.53897525  0.28954366  0.66643144 -0.09009203  0.57269675
## [26]  0.04207847 -0.29460512  0.06032883 -0.05954098  0.21969730
```

Sprawdzenie, które liczby są większe bądź równe 0 jest możliwe poprzez zasto-

sowanie komendy:

```
dane2 >= 0
```

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

Otrzymany wynik 30-tu wartości (TRUE/FALSE) wskazuje czy w danym indeksie wartość spełnia dany warunek logiczny. Jeśli chcemy się pozbyć wartości ujemnych możemy np. stworzyć wektor wartości logicznych, zapisać go jako nowy obiekt i podstawić jako indeks:

```
indeks <- dane2 >= 0 # tworzymy wektor indeksujący
dane2 [ indeks ]
```

```
## [1] 0.18754995 0.37016365 0.39507023 0.91508907 0.25714506 0.82342568
## [7] 0.55519429 0.98477907 0.17569978 0.79882611 0.88125978 0.28954366
## [13] 0.66643144 0.57269675 0.04207847 0.06032883 0.21969730
```

Lub z pominięciem etapu tworzenia tymczasowego obiektu indeksującego:

```
dane2 [ dane2 >= 0 ]
```

```
## [1] 0.18754995 0.37016365 0.39507023 0.91508907 0.25714506 0.82342568
## [7] 0.55519429 0.98477907 0.17569978 0.79882611 0.88125978 0.28954366
## [13] 0.66643144 0.57269675 0.04207847 0.06032883 0.21969730
```

3.4.2 Funkcja *which*

W wielu przypadkach bardziej wygodna w użyciu będzie funkcja `which()` (ang. “który”) zwracająca numery pozycji spełniających dany warunek logiczny.

`which()` (ang. “który/które”) - czyli które elementy spełniają dany warunek logiczny.

Poniższa komenda wskazuje indeksy wektora `dane2` większe od 0

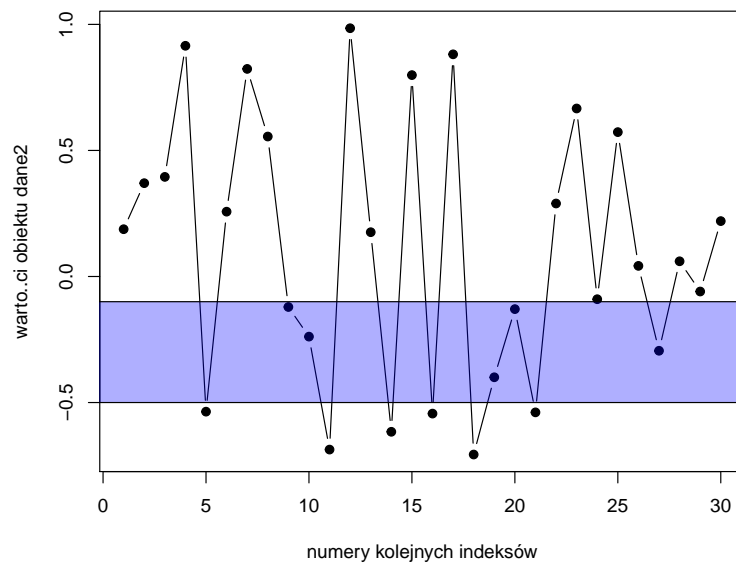
```
which(dane2>0)
```

```
## [1] 1 2 3 4 6 7 8 12 13 15 17 22 23 25 26 28 30
```

Możemy także jednorazowo wykonać 2 testy logiczne, takby aby wybrać np. liczby które są jednocześnie większe bądź równe -0.5 i mniejsze bądź równe -0.1. Graficznie taką zależność możemy wyobrazić sobie jako znalezienie elementów, które zawierają się w niebieskim prostokącie.

```
## Warning in title(...): conversion failure on 'wartości obiektu dane2' in
## 'mbcsToSbcs': dot substituted for <c5>
```

```
## Warning in title(...): conversion failure on 'wartości obiektu dane2' in
## 'mbcsToSbcs': dot substituted for <9b>
```



Rysunek 3.2: Znajdowanie liczb znajdujących się w przedziale domkniętym od -0.5 od -0.1.

Odszukanie indeksów tych elementów w składni **R** jest możliwe z wykorzystaniem wyrażeń logicznych i funkcji `which()`

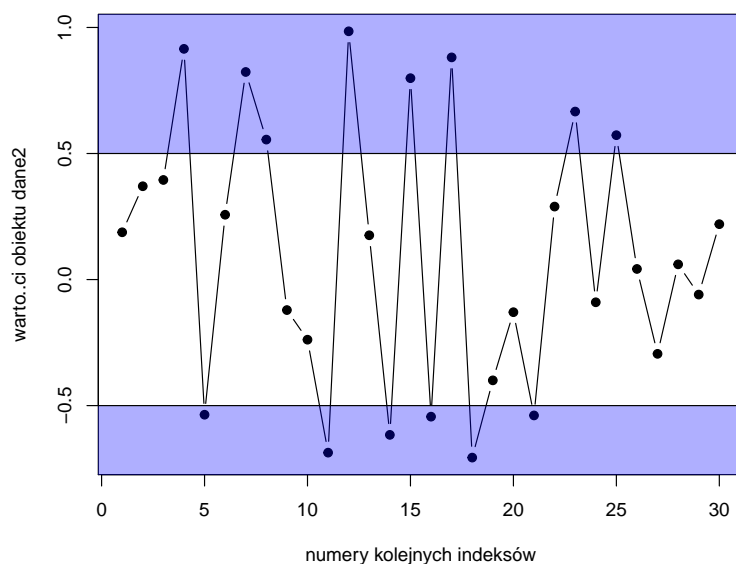
```
which(dane2 >= -0.5 & dane2 <= -0.1 )
```

```
## [1]  9 10 19 20 27
```

Operator `&` oznacza iloczyn logiczny zbioru. Jeśli interesowałyby nas wartości, które są jednocześnie mniejsze od -0.5 LUB są większe od 0.5 (jak na rys. 3.3), wówczas zamiast iloczynu logicznego powinniśmy użyć sumy logicznej zbioru (czyli operatora `|`).

```
## Warning in title(...): conversion failure on 'wartości obiektu dane2' in
## 'mbcsToSbcs': dot substituted for <c5>
```

```
## Warning in title(...): conversion failure on 'wartości obiektu dane2' in
## 'mbcsToSbcs': dot substituted for <9b>
```



Rysunek 3.3: Znajdowanie wartości mniejszych od -0.5 i większych od 0.5

```
which(dane2< -0.5 | dane2> 0.5)
```

```
## [1] 4 5 7 8 11 12 14 15 16 17 18 21 23 25
```

Jeśli interesują nas same wartości spełniające dany warunek logiczny (a nie indeksy) możemy zapisać wynik powyższego działania do nowego obiektu i wykorzystać go jako wektor indeksujący

```
indeks <- which(dane2< -0.5 | dane2> 0.5)
dane2[indeks]
```

```
## [1] 0.9150891 -0.5361438 0.8234257 0.5551943 -0.6865552 0.9847791
## [7] -0.6160373 0.7988261 -0.5438531 0.8812598 -0.7062056 -0.5389752
## [13] 0.6664314 0.5726967
```

Jeśli chcemy wyświetlić wszystkie pozostałe wartości możemy postawić znak minus w [], co da nam wartości indeksu pomnożone razy -1

```
dane2[-indeks] # lub to samo zapisane jako: dane2[indeks * -1]

## [1] 0.18754995 0.37016365 0.39507023 0.25714506 -0.12100064
## [6] -0.23853956 0.17569978 -0.39978640 -0.12945671 0.28954366
## [11] -0.09009203 0.04207847 -0.29460512 0.06032883 -0.05954098
## [16] 0.21969730
```

3.5 Zadania podsumowujące

3.5.1 Skrypt

1. Utwórz nowy skrypt i skopiuj do niego zawartość kodu ze strony: [<http://enwo.pl/przetwarzanie/lorenz.r>] (<http://enwo.pl/przetwarzanie/lorenz.r>)
2. Uruchom skrypt

3.5.2 Tworzenie wektorów

1. Stwórz wektor `miasta` składający z 3-ech losowych nazw miejscowości.
2. Dołącz do obiektu `miasta` kolejne o nazwie `Wzdół Rządowy` [1], [2].
3. Nadpisz zawartość wektora `miasta` usuwając ostatnią nazwę, gdyż `Wzdół Rządowy` jest wsią a nie miastem.

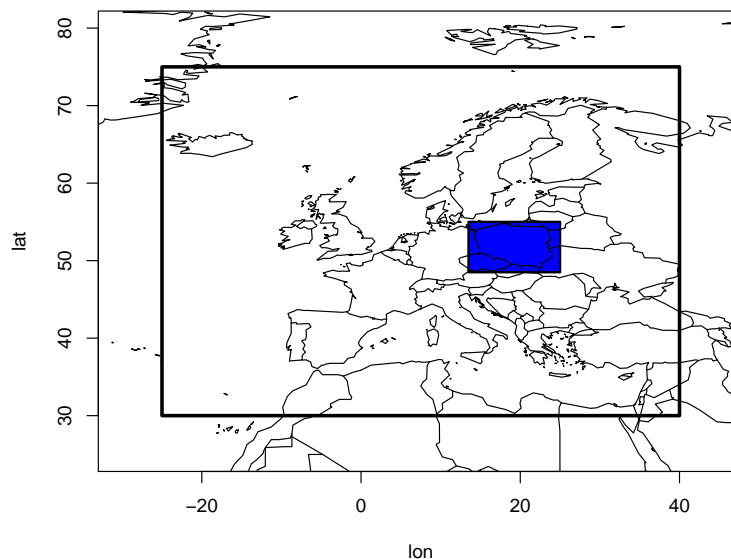
3.5.3 Indeksowanie

1. Wyświetl zawartość wbudowanej zmiennej `letters`
2. Jaka jest 24-ta litera alfabetu zapisanego w obiekcie `letters`?
3. Ile liter jest dalej w alfabecie niż litera `s` ? Wykorzystaj zapytanie logiczne `>`

Przydatna może być także wiedza o wartościach numerycznych dla `TRUE/FALSE` oraz funkcji `sum()`.
4. Korzystając z operatora `[]` wyświetl:

- a) parzyste litery alfabetu (2,4,6 ...)
- b) nieparzyste litery alfabetu (1,3,5 ...)

5.1. Stwórz obiekty `lon` i `lat` zawierające wartości południków (`lon`) i równoleżników (`lat`) dla obszaru Europy odpowiadającej siatce modelu GFS. Model GFS ma rozdzielczość przestrzenną 0.25×0.25 stopnia. Przyjmij roboczo, że obszar Europy zawiera się w od 25W do 40E oraz od 30N do 75N (rys. 3.4). Dla oznaczenia długości geograficznej przyjmij wartości ujemne dla długości geograficznej zachodniej (tj. 25W = -25) i dodatnie dla długości geograficznej wschodniej (tj. 40E = 40). Kontrolnie możesz sprawdzić długość powstałych obiektów (funkcją `length()`). Sprawdź czy stworzono 261 południków i 181 równoleżników.



Rysunek 3.4: Schemat domeny obliczeniowej modelu GFS dla Europy

5.2. Za pomocą funkcji `which()` wskaż numery indeksów `lon` i `lat`, które zawierają obszar Polski (3.4). Przyjmij roboczo, że długości geograficzne powinny zawierać się w przedziale od 13.5 do 25, a szerokości od 48.5 do 55.

5.3. Z obiektów `lon` i `lat` wydobądź poprzez operator `[]` wartości południków i równoleżników w podanym powyżej zakresie (13.5E-25.0E, 48.5N-55.0N). Wartości zapisz do nowych obiektów `lon2` i `lat2`.

5.4. Ile elementów zawierają nowe obiekty `lon2` i `lat2` ?

Zadanie domowe

1. Korzystając z wbudowanego systemu pomocy puść “totolotka” za pomocą funkcji `sample()` generując zestaw liczb od 1 do 49, a następnie wywołując parametr `size=` dla sześciu liczb.
2. Wygeneruj i zapisz w postaci wektora dowolny ciąg liczbowy składający się z 20-50 liczb. Następnie korzystając z wygenerowanych liczb i wbudowanego systemu pomocy przetestuj działanie poniższych funkcji:

Tabela 3.2: Wybrane funkcje uzupełniające niezbędną wiedzę do pracy na wektorach w **R**

Nazwa funkcji	Działanie
<code>summary()</code>	statystyki podsumowujące
<code>range()</code>	zwraca jednocześnie minimalną i maksymalną wartość
<code>rev()</code>	odwraca kolejność elementów
<code>rep()</code>	replikacja / powtarzanie / zwielokrotnienie
<code>sort()</code>	sortowanie
<code>length()</code>	długość obiektu

Zadania sprawdzające

<https://goo.gl/forms/EMbuMMHVSIkQeP2q2>

Rozdział 4

Ramki danych i macierze

W poprzednim rozdziale omówiono kilka schematów posługiwania się wektorami, który są jednocześnie najbardziej elementarnym sposobem przetwarzania danych w **R**.

4.1 Ramki danych

Przy analizie danych (w tym także danych meteorologicznych) wykorzystywane są inne niż wektory schematy przechowywania i przetwarzania danych. Najczęściej są to tzw. ramki danych (ang. *data frame*), które na pierwszy rzut oka przypominają tradycyjną “tabelkę z Excela”.

Najważniejsze cechy ramki danych wypisano poniżej:

1. Każda ramka danych powinna zawierać wartości uporządkowane w kolumnach.
2. Każda z kolumn jest wektorem i musi mieć taką samą długość.
3. Różne kolumny mogą przechowywać różne typy danych.

Poniżej zamieszczono fragment ramki danych zawiera dane pomiarowe ze stacji Wojewódzkiego Inspektoratu Ochrony Środowiska - Poznań-Polanka. W kolumnie *date* zawarto dane w typie umożliwiającym przechowywanie czasu, w kolejnych kolumnach umieszczono wartości koncentracji PM10, PM2.5, temperatury

powietrza, prędkości wiatru o oraz kierunku wiatru.

```
##           date PZ_Pol_pm10 PZ_Pol_pm25 PZ_T2M PZ_WS PZ_WD
## 95034 2016-11-07 17:00:00          24          16      4      1    278
## 95035 2016-11-07 18:00:00          22          17      4      1    287
## 95036 2016-11-07 19:00:00          22          17      3      1    322
## 95037 2016-11-07 20:00:00          23          16      3      1    326
## 95038 2016-11-07 21:00:00          24          16      3      0    307
## 95039 2016-11-07 22:00:00          24          16      2      0    303
```

4.2 Praca na ramkach danych

Do nauki pracy na ramkach danych wykorzystamy domyślnie wgrane zbiór danych o nazwie *airquality*. Wczytanie przykładowego zbioru danych wgranego wraz z **R** jest możliwe poprzez funkcję `data()`. Nasz zbiór nazywa się *airquality*. Szczegóły dotyczące analizowanego zbioru danych dostępne są po wydaniu komendy `?airquality` lub `"airquality"`

Jeśli chcemy wczytać i wyświetlić pierwsze kilka rzędów naszej ramki danych spróbujmy najpierw ją wczytać do pamięci komputera a następnie wyświetlić pierwsze lub ostatnie 6 wierszy za pomocą funkcji `head()` lub `tail()`

```
data("airquality")
head(airquality) # funkcja head() wyświetla pierwsze 6 wartości
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

Po wczytaniu ramki danych powinna być ona dostępna w prawym górnym rogu środowiska RStudio w zakładce *Environment*. Możesz kliknąć w tabelkę widoczną obok nazwy ramki danych i wyświetlić jej zawartość w postaci graficznej. Klikając na nazwę kolumny możesz także automatycznie posortować bieżący wi-

dok rosnąco/malejąco względem wartości w danej kolumnie. Standardowo, po wczytaniu zbioru danych można użyć funkcję `summary()` wyświetlającą podsumowanie statystyczne wartości w poszczególnych kolumnach ramki danych.

4.2.1 Jak odnosić się do elementów ramki?

4.2.1.1 Odniesienie przez nazwę

Wyboru kolumny z ramki danych dokonuje się za pomocą operatora `$` poprzedzonego nazwą ramki danych a po znaku `$` wpisuje się nazwę kolumny. Jeśli nie jesteśmy pewni jakie są nazwy kolumn w naszej ramce danych można je zawsze sprawdzić za pomocą funkcji `colnames()` lub `names()`. Pamiętaj także o stosowaniu tabulatora!

Przykładowo, jeśli chcemy pobrać do analizy całą kolumnę z wartościami temperatur (*Temp*) z ramki danych `airquality` to komenda powinna wyglądać następująco:

```
airquality$Temp
```

```
##      [1] 67 72 74 62 56 66 65 59 61 69 74 69 66 68 58 64 66 57 68 62 59 73 61
##     [24] 61 57 58 57 67 81 79 76 78 74 67 84 85 79 82 87 90 87 93 92 82 80 79
##     [47] 77 72 65 73 76 77 76 76 76 75 78 73 80 77 83 84 85 81 84 83 83 88 92
##     [70] 92 89 82 73 81 91 80 81 82 84 87 85 74 81 82 86 85 82 86 88 86 83 81
##     [93] 81 81 82 86 85 87 89 90 90 92 86 86 82 80 79 77 79 76 78 78 77 72 75
##    [116] 79 81 86 88 97 94 96 94 91 92 93 93 87 84 80 78 75 73 81 76 77 71 71
##    [139] 78 67 76 68 82 64 71 81 69 63 70 77 75 76 68
```

Zadania kontrolne

1. Wiedząc, że wynikiem pobrania kolumny z ramki danych jest wektor oblicz średnią temperaturę powietrza zbioru `airquality` w Fahrenheitach oraz stopniach Celsjusza ($T[*C] = (T[*F]-32)*(5/9)$)
2. Oblicz minimalną prędkość wiatru z pierwszych 20-tu pomiarów
3. Utwórz histogram temperatur (funkcja `hist()`)

4.2.1.2 Odniesienie przez indeks

Praca na wartościach przechowywanych w ramach danych możliwa jest także poprzez indeksowanie operatorem `[]`, którego używaliśmy poprzednio przy wektorach. Jedyna różnica polega na tym, że w nawiasie kwadratowym należy zadeklarować 2 wektory określające położenie wierszy i kolumn oddzielone przecinkiem.

Przed przecinkiem należy wskazać indeksy wierszy, a po przecinku indeksy kolumn. Jeżeli pole indeksujące wierszy lub kolumn pozostanie puste wówczas zostają wybrane wszystkie elementy.

Jeśli chcemy wybrać wszystkie wartości z kolumny `Temp` wówczas musimy wiedzieć, że jest to 4-ta kolumna. Jako, że chcemy wybrać wszystkie rzędy to równoważnik dla komendy `airquality$Temp` to: `airquality[,4]`.

Jeśli chcemy wyświetlić cały pierwszy rząd to polecenie powinno wyglądać następująco:

```
airquality[1,]
```

```
##      Ozone Solar.R Wind Temp Month Day
## 1      41      190  7.4   67      5   1
```

Można także wskazać dowolne wiersze, np. od 10-ego do 15-ego w odwrotnej kolejności i tylko wybrane kolumny, np. dla ozonu i wiatru (1-sza i 3-cia). Istotne jest tylko podanie odpowiedniego wektora wartości w odpowiednich miejscach:

```
airquality[15:10,c(1,3)]
```

```
##      Ozone Wind
## 15      18 13.2
## 14      14 10.9
## 13      11  9.2
## 12      16  9.7
## 11       7  6.9
## 10      NA  8.6
```

Zadania kontrolne

1. Ze zbioru `airquality` wybierz za pomocą jednego polecenia wiersze: 5-10,

15-17 i 100-102 oraz wszystkie kolumny

2. Korzystając z indeksów ujemnych (tzw. wybór przez negację pominię pierwsze 3 kolumny i pierwsze 100 wierszy)
3. Korzystając z wyrażeń logicznych lub funkcji `which()` wyświetl jedynie wiersze, w których temperatura powietrza była wyższa niż 90°F

4.2.2 Tworzenie ramek danych

Tworzenie i modyfikacje ramek danych jest możliwe przynajmniej na kilka sposobów. Najbardziej podstawowym jest zastosowanie funkcji `data.frame()`, która jako argumenty przyjmuje wartości wektorów o równych długościach. Przykładowy fragment kodu stworzy nam ramkę danych nazwaną `ramka` zawierającą 3 nazwane kolumny i 5 wierszy:

```
ramka <- data.frame(literki=letters[1:5], cyferki=1:5, losowe=runif(5))
ramka
```

```
##   literki cyferki    losowe
## 1      a      1 0.54664830
## 2      b      2 0.99648647
## 3      c      3 0.93739406
## 4      d      4 0.02059932
## 5      e      5 0.53162818
```

4.2.2.1 Modyfikacja ramek danych

Dane przechowywane w ramkach danych można modyfikować. Jeśli zmiana ma dotyczyć pojedynczej wartości to należy stworzyć komendę zwracającą nam tę wartość i nadpisać do niej nową wartość operatorem przypisania. Przykładowo, modyfikacja pierwszej wartości temperatury powietrza ze zbioru `airquality` możliwa jest w poniższy sposób:

```
airquality$Temp[1] # wyświetlamy wartosc pierwotna
```

```
## [1] 67
```

```
airquality$Temp[1] <- 100 # nadajemy nowa wartosc
head(airquality) # wyswietlamy dla pewnosci pierwsze 6 rzadow
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4  100     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

Często do istniejącej ramki danych chcemy dopisać wyniki obliczeń jako nową kolumnę. W **R** łączenie odpowiednich obiektów w ramki danych po kolumnach jest możliwe dzięki funkcji `cbind.data.frame()`, natomiast do łączenia po wierszach służy funkcja `rbind.data.frame()`.

Do utworzenia nowej kolumny szybszym rozwiązaniem w wielu przypadkach jest po prostu wykorzystanie do tego celu operatora przypisania `<-` wraz z odwołaniem do istniejącej ramki danych i nazwą nowej kolumny. Przeliczmy zatem temperaturę z Fahrenheitów na stopnie Celsjusza i zapiszmy je w kolumnie `TempC`:

```
# przeliczamy temperature z Fahrenheitow na Celsjusze i zapisujemy do nowej
# kolumny 'TempC'
airquality$TempC <- (airquality$Temp-32)*(5/9)
head(airquality) # wyswietlmy pierwsze 6 rzadow dla pewnosci
```

```
##   Ozone Solar.R Wind Temp Month Day   TempC
## 1    41     190  7.4  100     5   1 37.77778
## 2    36     118  8.0   72     5   2 22.22222
## 3    12     149 12.6   74     5   3 23.33333
## 4    18     313 11.5   62     5   4 16.66667
## 5    NA      NA 14.3   56     5   5 13.33333
## 6    28      NA 14.9   66     5   6 18.88889
```

4.3 Macierze

Oprócz ramek danych bardzo często w zastosowaniach GIS lub na potrzeby tzw. reanaliz meteorologicznych stosowane są dane zapisane w formie wielowymiarowych macierzy. Macierze w najprostszej postaci (2-wymiarowej) to po prostu tabele z danymi, wizualnie bardzo podobne do ramek danych, przy czym nie posiadają one nazw kolumn i nie można się do nich odnosić poprzez nazwę a jedynie poprzez `[]`.

Tworzenie macierzy wielowymiarowych odbywa się poprzez funkcję `array()` a w postaci 2-wymiarowej prostsza w użyciu jest funkcja `matrix()` (w dosłownym tłumaczeniu macierz). Za pomocą opcji `nrow` oraz `ncol` można zadeklarować liczbę wierszy i kolumn tworzonej macierzy. Przykładowo:

```
matrix(1:12, nrow=3) # tworzymy macierz z wartościami od 1 do 12 w 3-ech rzędach
```

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

```
matrix(1:12, ncol=3) # tworzymy macierz z wartościami od 1 do 12 w 3-ech kolumnach
```

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

Zwróć uwagę, że poszczególne elementy są użyte do wypełnienia w pierwszej kolejności 1-szej kolumny, potem 2-giej, itd. Jeśli chcemy użyć danych do wypełnienia po rzędach, musimy komputer o tym poinformować za pomocą opcji `byrow=T`:

```
m2<-matrix(1:12,ncol=3,byrow=T)
```

```
m2
```

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

```
## [3,]    7    8    9
## [4,]   10   11   12
```

Tworzenie macierzy można wykonywać np. poprzez złączenie kolumn lub rzędów o jednakowej długości. W zależności od tego w jaki sposób chcemy macierz utworzyć wykorzystujemy funkcje `cbind()`, (skrót od *column bind*) lub `rbind()` (skrót od *row bind*):

```
x <- c(1,3,2,10,5) # tworzymy wektor 'x'
x

## [1]  1  3  2 10  5

y <- 1:5 # tworzymy wektor 'y'
m1<-cbind(x,y) # łączymy wektory po kolumnach i tworzymy macierz 'm1'
m1 # wyświetlamy powstała macierz
```

```
##      x y
## [1,]  1 1
## [2,]  3 2
## [3,]  2 3
## [4,] 10 4
## [5,]  5 5
```

Analogicznie można utworzyć macierz łącząc wektory po wierszach za pomocą funkcji `rbind()`:

```
m2 <- rbind(x,y) # łączymy po wierszach i tworzymy nowa macierz 'm2'
m2

##      [,1] [,2] [,3] [,4] [,5]
## x      1    3    2   10    5
## y      1    2    3    4    5
```

Możemy także uzyskać podstawowe informacje oraz wykonać podstawowe operacje matematyczne na macierzach:

```
dim(m1) # podaje wymiary macierzy
```

```
## [1] 5 2
```



```
t(m1) # transpozycja
```

```
##      [,1] [,2] [,3] [,4] [,5]
## x      1      3      2     10      5
## y      1      2      3      4      5
```

```
5.2*m1 # iloczyn skalarny
```

```
##           x      y
## [1,]   5.2  5.2
## [2,]  15.6 10.4
## [3,]  10.4 15.6
## [4,]  52.0 20.8
## [5,]  26.0 26.0
```

```
m1+m1 # dodawanie macierzy
```

```
##           x      y
## [1,]      2      2
## [2,]      6      4
## [3,]      4      6
## [4,]     20      8
## [5,]     10     10
```

```
m1*m1 # mnożenie analogicznych elementów
```

```
##           x      y
## [1,]      1      1
## [2,]      9      4
## [3,]      4      9
## [4,]     100     16
## [5,]     25     25
```

4.3.1 Indeksowanie macierzy

Wyciąganie poszczególnych elementów z macierzy wymaga zadeklarowania jej wszystkich wymiarów:

```

m2<-matrix(c(1,3,2,5,-1,2,2,3,9),ncol=3,byrow=T);m2

##      [,1] [,2] [,3]
## [1,]    1    3    2
## [2,]    5   -1    2
## [3,]    2    3    9

# tworzymy macierz i ją wyświetlamy

m2[2,3] #element macierzy m2 w 2-gim rzędzie i 3-ciej kolumnie

## [1] 2

m2[2,] #cały 2-gi rząd

## [1]  5 -1  2

m2[,3] #cała 3-cia kolumna

## [1] 2 2 9

m2[-1,] #cała macierz m2 bez pierwszego rzędu

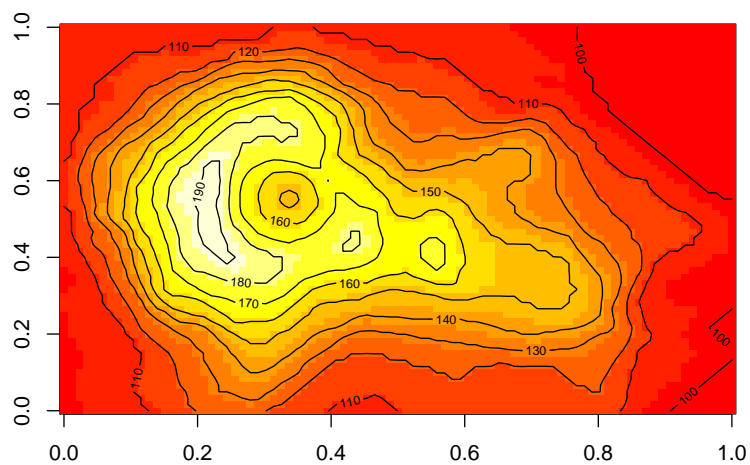
##      [,1] [,2] [,3]
## [1,]    5   -1    2
## [2,]    2    3    9

m2[, -1] #i bez pierwszej kolumny

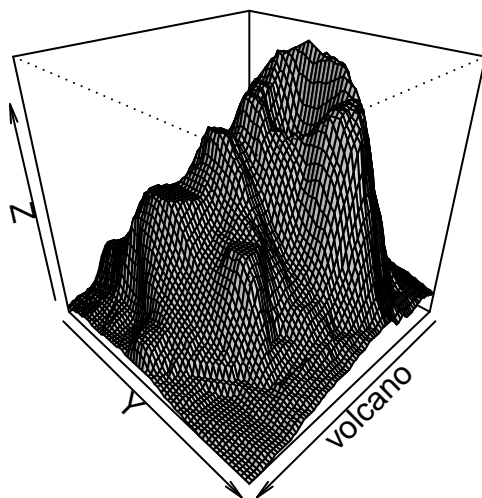
##      [,1] [,2]
## [1,]    3    2
## [2,]   -1    2
## [3,]    3    9
...

```

Macierze w naukach atmosferycznych są stosowane przede wszystkim do nieco bardziej skomplikowanych obliczeń oraz do wizualizacji danych. Przetestujmy to drugie zastosowanie wczytując zbiór danych `volcano` zawierający cyfrowy model terenu pewnego wulkanu i zwizualizujemy go za pomocą funkcji `image()` oraz `contour()` lub jako model 3d z użyciem funkcji `persp()`. Postaraj się uzyskać poniższy efekt:



Rysunek 4.1: Wulkan - rzut z góry



Rysunek 4.2: Wulkan - rzut perspektywiczny

Zadanie domowe

1. Wykonaj przykłady z podręcznika P. Biecka dotyczące ramek danych i macierzy
2. Przeczytaj z ww. podręcznika rozdział dotyczący list i ogólnego schematu pracy na danych tego typu

Zadania sprawdzające

<https://goo.gl/forms/QkiVqWliDJlrhLAQ2>

Rozdział 5

Wczyt i zapis danych

Elementarną umiejętnością pracy na danych w **R** jest możliwość zaczytania wybranego/dowolnego formatu oraz dalsza praca już na danych dostępnych w środowisku **R/RStudio**. Prawdopodobnie dotychczas dane zapisywałeś(aś) w formatach arkuszy kalkulacyjnych (xls/xlsx/ods). Tymczasem w naukach atmosferycznych jest to w rzeczywistości jeden z najrzadziej wykorzystywanych formatów...

R obsługuje bardzo szeroką gamę formatów danych, a te które nie są natywnie wspierane zwykle mogą być zaimportowane do **R** za pomocą bezpłatnych bibliotek. W ten sposób w **R** można jednocześnie pracować w jednym środowisku na różnych danych, bez konieczności *przełączania* się za każdym razem do programu obsługującego dany format lub posiadającego dane możliwości (np. pobranie danych -> przygotowanie danych przestrzennych -> wizualizacja w postaci mapy).

W naukach atmosferycznych najczęściej stosowane formaty danych obsługiwane przez **R** to:

- Dane przechowywane w formie “tabelarycznej” (np.: `csv`, `ASCII`, `xls/xlsx`, `GoogleSheets` oraz w postaci bazodanowej (np. `SQL` i pochodne)).
- Formaty danych specjalistycznych GIS: (np. `HDF4`, `HDF5`, `GeoTIFF`, `Shapefile`, `ASC` i innych wspieranych przez bibliotekę `GDAL`).

- Specjalistyczne formaty danych meteorologicznych (np. GRIB, NetCDF-3, NetCDF-4).

5.1 Katalog roboczy

Zanim przejdziemy do wczytywania danych musimy zaznajomić się ze sposobem ustalania katalogów roboczych **R**. Domyślnie po uruchomieniu **R** pracujemy w katalogu domowym, który może być zlokalizowany w różnych miejscach dysku. Aby sprawdzić obecny katalog roboczy (ang. *Workind directory*) należy posłużyć się funkcję `getwd()` bez żadnych dodatkowych argumentów

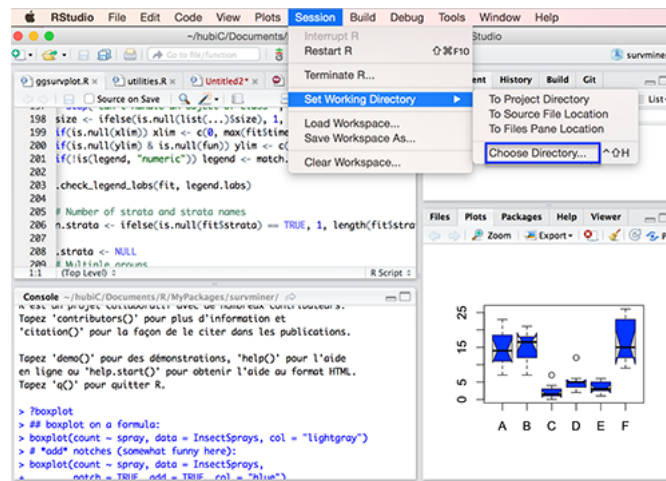
```
getwd()
```

```
## [1] "/home/bartosz/github/przetwarzanie"
```

Jeśli chcemy zmienić katalog roboczy możemy to zrobić na 3 różne sposoby:

1. Użyć funkcji `setwd()` podając ścieżkę do wybranego katalogu (wersja zaawansowana)
2. Wybrać z górnego menu **RStudio** opcję **Session**, następnie **Set working directory** i następnie **Choose directory**, analogicznie do zrzutu na poniższym ekranie. Ważne aby ustawić jedynie KATALOG a nie PLIK w którym będziemy pracować i przechowywać nasze dane. Po poprawnym wskazaniu katalogu roboczego w konsoli pojawi się wynik działania jako wywołana funkcja `setwd()` ze wskazanym katalogiem.
3. Analogiczny efekt jak w pkt. 2 można uzyskać za pomocą skrótu klawiszowego `ctrl+shift+h`

W tym momencie wszystkie pliki znajdujące się naszym katalogu mogą być od razu znalezione przez **R** (tj. bez wpisywania pełnej ścieżki do pliku). Jeśli chcesz sprawdzić jakie to są pliki zawsze możesz użyć funkcji `dir()` wyświetlającej zawartość obecnego katalogu roboczego `dir()`.



Rysunek 5.1: Wybór katalogu roboczego w RStudio

5.2 Wczyt danych

Dane do ćwiczeń pobierz ze strony: <http://www.enwo.pl/przetwarzanie/dane> i zapisz je na Pulpicie w folderze `dane` lub w innej znanej lokalizacji. Następnie ustaw katalog roboczy na odpowiedni folder z pobranymi plikami. Z poziomu konsoli **R** sprawdź zawartość katalogu poleceniem `dir()`. W wyniku powinieneś otrzymać następujące pliki:

5.2.1 Format tekstowy (CSV i pokrewne)

Naukę wczytywania danych rozpoczniemy od danych tabelarycznych zapisanych w formacie tekstowym. Pomimo wielu technicznych aspektów powodujących problemy, jest to w dalszym ciągu najbardziej uniwersalny format wymiany i zapisu danych. Najczęściej pliki przechowujące dane w takim kształcie mają rozszerzenie `.csv` od wyrażenia *comma separeted value* (ang. *wartości rozdzielone przecinkami*), `.tsv` (wartości rozdzielone znakami tabulacji), `.asc` lub po prostu `.txt`. Należy przy tym zaznaczyć, że rozszerzenie pliku nie jest przez **R** brane pod uwagę i dlatego użytkownik musi precyzyjnie zadeklarować wszystkie ustawienia związane z wczytem danych!

Do wczytywania danych tekstowych najczęściej stosowana jest funkcja:

`read.table()` lub w nieco zmodyfikowanej wersji funkcja `read.csv()`. Obie funkcje są bardzo podobne jeśli chodzi o sposób użycia i różnią się na ogół domyślnymi wartościami argumentów. Przed próbą wczytania pliku tekstowego do **R** należy w pierwszej kolejności zapoznać się z jego strukturą. W tym celu otwórz plik `pl1.csv` za pomocą notatnika lub innego edytora tekstowego i zapisz na kartce:

- czy plik ma nazwy kolumn/wierszy?
- czy dane w poszczególnych kolumnach (poza nazwami kolumny/wierszy) są tylko typu liczbowego?
- jaki znak jest separatorem miejsc dziesiętnych?
- jaki znak rozdziela kolejne pola?
- czy braki danych są zapisane w inny sposób niż poprzez wartości NA (częstym standardem jest stosowanie np. zapisu -9999, -99.99, —, pozostawianie pustego pola, itp.)?

Te informacje są niezbędne do ustawienia poszczególnych argumentów funkcji `read.table()` lub `read.csv()`. Najważniejsze z nich:

- **file** - pełna ścieżka do pliku lub po prostu nazwa pliku (jeśli znajduje się w katalogu roboczym), możliwe jest także podanie adresu internetowego
- **header** - czy plik posiada nagłówek (podpisane nazwy kolumn) w pierwszym wierszu? Domyślnie wartość tego parametru jest ustawiona na **TRUE**
- **sep** - separator kolumn - jaki znak rozdziela wartości w poszczególnych kolumnach. Może być to średnik (“;”), przecinek (“,”), spacja (“ ”), tabulator (`\t`), lub inny dowolnie zadeklarowany znak w cudzysłowie
- **dec** - separator miejsc dziesiętnych. Analogicznie jak dla opcji separatora kolumn
- **na.strings** - sposób zapisu i traktowania braków danych
- **stringsAsFactors** - czy wartości tekstowe mają być traktowane jako typ czynnikiowy (domyślnie **TRUE**, ale zwykle warto zmienić na **FALSE**)
- ... reszta opcji dostępna we wbudowanej pomocy funkcji `read.table()`)

Znając powyższe argumenty i po zaznajomieniu się z wbudowaną pomocą spróbuj użyć funkcji

`read.csv()` do wczytu przeglądanego pliku. Plik ma rozszerzenie `.csv` i jest rozdzielony spacjami, więc intuicyjnie rzecz biorąc powinna być to najlepsza opcja. W razie konieczności otwórz okno pomocy dla testowanej funkcji.

[illegible]

```
## 2 -0.01
## 3 -0.72
## 4  2.73
## 5  0.88
## 6 -1.24
## 7 -0.38
## 8 -3.03
## 9  2.14
## 10 -0.12
## 11 -3.20
## 12  1.22
## 13 -0.64
## 14 -1.09
## 15  2.16
## 16  0.02
## 17  0.76
## 18  1.10
## 19  1.16
## 20 -0.18
## 21 -0.93
## 22 -0.54
## 23  1.97
## 24  1.38
## 25 -4.44
## 26 -4.88
## 27  0.41
## 28 -1.69
## 29  0.91
## 30  1.92
```

Wygląda na to, że wszystko wczytało się poprawnie. Samo wyświetlenie zawartości pliku nie pozwala na dalszą pracę z danymi w **R**. Musimy wynik polecenia wczytać do zmiennej (za pomocą `<-`, `=` lub `->`) aby był widoczny zakładce ‘Environment’ pod wskazaną nazwą. Nazwijmy ten obiekt ‘dane’ i przejrzymy jego zawartość za pomocą graficznej przeglądarki ramek danych lub fragmentarycznie w konsoli.

```
dane <- read.csv("p11.csv")
head(dane)
```

```
##      rok      I      II     III     IV      V      VI      VII     VIII     IX      X      XI
## 1 1971 -2.96  0.37 -0.11  7.41 14.48 14.91 17.81 18.74 11.25 8.39 2.61
## 2 1972 -5.81  0.27  3.91  7.36 12.48 16.25 19.40 16.56 11.36 6.14 4.36
## 3 1973 -1.67  1.27  3.69  6.13 12.35 15.77 17.58 17.12 13.14 6.62 1.85
## 4 1974  0.06  2.29  4.37  6.89 10.75 14.04 15.62 17.70 13.40 6.29 3.86
## 5 1975  2.97 -0.56  3.87  6.58 13.35 15.40 18.55 18.27 15.79 7.98 1.75
## 6 1976 -1.97 -3.32 -0.85  6.72 11.95 15.09 18.06 15.31 12.58 7.59 4.71
##      XII
## 1  3.02
## 2 -0.01
## 3 -0.72
## 4  2.73
## 5  0.88
## 6 -1.24
```

Usunięcie zmiennej

Każdy obiekt zaczytany do pamięci operacyjnej widoczny w prawym górnym rogu w zakładce **Environment** zmniejsza dostępną pamięć dla następnych obiektów. Jeśli chcemy się pozbyć wszystkich zadeklarowanych obiektów możemy kliknąć w symbol pędzelka w prawym górnym rogu i potwierdzić chęć usunięcia wszystkich obiektów.

W przypadku gdy chcemy usunąć tylko wskazane obiekty możemy posłużyć się funkcją `rm()` wpisując jako argument nazwę obiektu (bez cudzysłowia), którego chcemy się pozbyć. Jeśli pracujemy w środowisku **R** (a nie **RStudio**) to nazwy bieżących zmiennych możemy wyświetlić za pomocą funkcji `ls()`.

Zadania kontrolne

1. Na podstawie wczytanego zbioru danych oblicz średnią roczną temperaturę powietrza w roku 1971 i 2000
2. Oblicz średnią temperaturę stycznia i grudnia w wieloleciu 1971-2000

3. Sprawdź w sytemie pomocy działanie funkcji `boxplot()`. Następnie zastosuj ją do wczytanego zbioru danych. 3.1. Wywołaj jeszcze raz funkcję `boxplot()` ale bez wartości dla pierwszej kolumny (dla lat)
4. Wczytaj pozostałe zbiory danych tekstowych (tj. `pl2.tsv`, `pl3.csv`, `pl4.txt`) korzystając z funkcji `read.csv()` lub `read.table()`

5.2.1.1 Wczyt danych bezpośrednio z internetu

W odróżnieniu od pracy w arkuszach kalkulacyjnych **R** w wielu przypadkach nie wymaga wcześniejszego zapisu danych na dysku! Dane można wczytać bezpośrednio np. z internetu:

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

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

5.2.2 Pliki tekstowe o stałej szerokości kolumn

Zdarza się, że dane są zapisywane w formacie plików o odgórnie ustalonej szerokości plików (tzw. Fixed Width Format Files). Przykładowe dane w takim formacie umieszczono pod adresem <http://enwo.pl/przetwarzanie/dane/poz.txt>

Wczyt takiego formatu w **R** umożliwia funkcja `read.fwf()`, która oprócz wskazania nazwy pliku wymaga deklaracji argumentu `widths` w formie wektora ozna-

czającego szerokość poszczególnych kolumn. Przed wczytaniem pliku należy zatem dokładnie policzyć ile znaków (szerokości) zajmują pola poszczególnych kolumn. Wspomniany przed momentem plik z danymi meteorologicznymi dla Poznania zapiszmy na dysku. Powinien zawierać:

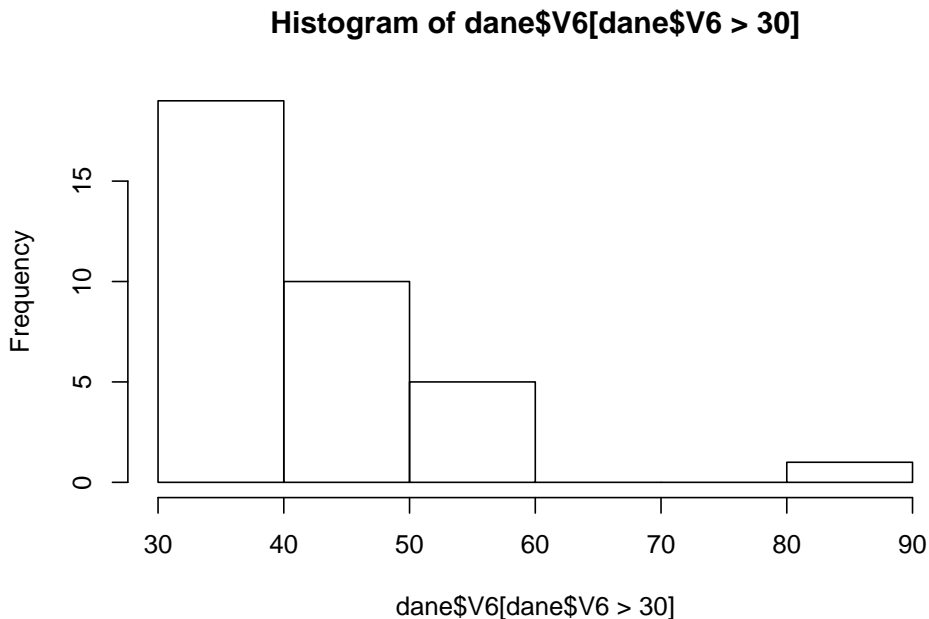
- 5 znaków dla kodu stacji
- 30 znaków dla nazwy stacji
- 4 znaki dla roku
- 2 znaki dla miesiący
- 2 znaki dla dni
- 8 znaków dla dobowych sum opadów atmosferycznych. *Brak opadu zapisany jest jako .09 w 3 ostatnich znakach pola*
- 7 znaków dla dobowej temperatury powietrza

Zatem pełna komenda wczytująca wspomniany plik oraz 2 komendy wyświetlające szybkie podsumowanie mogą wyglądać następująco:

```
dane <- read.fwf("poz.txt",widths = c(5,30,4,2,2,8,7), na.strings=".09")
summary(dane) # wyświetlenie
```

```
##          V1                      V2          V3
## Min.    :330   poznan           :17167   Min.    :1966
## 1st Qu.:330                                1st Qu.:1977
## Median :330                                Median :1989
## Mean    :330                                Mean    :1989
## 3rd Qu.:330                                3rd Qu.:2001
## Max.    :330                                Max.    :2012
##
##          V4          V5          V6          V7
## Min.    : 1.000   Min.    : 1.00   Min.    : 0.000   Min.    : -21.9
## 1st Qu.: 4.000   1st Qu.: 8.00    1st Qu.: 0.000   1st Qu.: 2.5
## Median : 7.000   Median :16.00   Median : 0.700   Median : 8.8
## Mean    : 6.523   Mean    :15.73   Mean    : 2.419   Mean    : 8.7
## 3rd Qu.:10.000   3rd Qu.:23.00   3rd Qu.: 2.900   3rd Qu.: 15.3
## Max.    :12.000   Max.    :31.00   Max.    :85.700   Max.    : 29.7
##
##          NA's    :6973
```

```
hist(dane$V6[dane$V6>30]) # histogram częstości występowania opadów > 30mm/dobę
```



5.2.3 Format Excela (.xls / .xlsx)

Dane pochodzące z arkuszy kalkulacyjnych zwykle są zapisywane w postaci plików `.xls` lub `.xlsx`. Nie jest to najbardziej “szczęśliwy” format do pracy w **R** ze względu na brak jednoznacznej interpretacji niektórych zapisów i formatowania (np. `dat`).

Jeśli mamy dane we wspomnianych formatach najczęściej stosuje się 2 rozwiązania:

1. Eksport danych z arkusza kalkulacyjnego do formatu tekstowego (np. `csv`)
2. Wykorzystanie bezpłatnych pakietów np. `readxl`, `XLconnect`, `openxlsx`, `gdata`, `xlsReadWrite`, `xlsx`...

Spróbujmy zastosować oba rozwiązania do plików `p15.xls` oraz `p16.xlsx` ...

5.2.4 Format binarny R

Najbardziej efektywnym sposobem przechowywania danych w **R** jest format binarny, który pozwala na szybki zapis i odczyt danych. Dane w tym formacie mają najczęściej rozszerzenie `.rda` lub `.Rdata` i mogą być wczytane za pomocą funkcji `load()`, przy czym w pliku może się znajdować więcej niż 1 obiekt, stąd też zmienne bezpośrednio są ładowane do środowiska bez możliwości deklarowania ich nazwy.

Spróbujmy pobrać przykładowe dane z adresu <http://www.enwo.pl/przetwarzanie/dane/pm10.Rdata> i zapiszmy je na dysku. Następnie wczytajmy je za pomocą funkcji `load()` i przejrzymy ich zawartość.

```
load("dane/pm10.Rdata")
head(ogrody_d)
```

```
##           date   yy mm dd   hh      ws      wd      pm10      air_t      slp
## 1 2005-01-01 2005   1   1 12.0 3.826087 275.5945 25.61364 0.2545455 1013.182
## 2 2005-01-02 2005   1   2 11.5 7.583333 253.0708 19.56250 0.3083333 1004.083
## 3 2005-01-03 2005   1   3 11.5 7.833333 267.5562 10.91667 0.1541667 1009.417
## 4 2005-01-04 2005   1   4 11.5 9.041667 271.8748 12.73913 0.4608696 1008.174
## 5 2005-01-05 2005   1   5 11.5 6.291667 260.1847 19.00000 2.1250000 1013.083
## 6 2005-01-06 2005   1   6 11.5 5.750000 259.3237 21.23913 2.8043478 1013.826
```

```
head(polanka_d)
```

```
##           date   yy mm dd   hh      ws      wd      pm10 pm25
## 1 2005-01-01 2005   1   1 12.0 3.826087 275.5945 21.543478  NA
## 2 2005-01-02 2005   1   2 11.5 7.583333 253.0708 12.291667  NA
## 3 2005-01-03 2005   1   3 11.5 7.833333 267.5562  9.666667  NA
## 4 2005-01-04 2005   1   4 11.5 9.041667 271.8748  8.395833  NA
## 5 2005-01-05 2005   1   5 11.5 6.291667 260.1847 18.978261  NA
## 6 2005-01-06 2005   1   6 11.5 5.750000 259.3237 19.875000  NA
```

Jeśli chcemy zaczytać plik bezpośrednio z internetu musimy funkcję `load()` o tym poinformować korzystając dodatkowo z funkcji `url()`:

```
load(url("http://biecek.pl/M00C/dane/koty_ptaki.rda"))
koty_ptaki
```

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

5.2.5 RDS

Jeśli chcemy pozostawić dowolność co do nazwy wczytywanego obiektu częstą alternatywą jest format `.rds`, który również jest wewnętrznym formatem środowiska **R**, ale przechowuje jeden obiekt, stąd też zwykle wczytuje się go do zmiennej. Funkcja do odczytu tego formatu to `readRDS()`. Warto zaznaczyć, że ta funkcja działa intuicyjnie jedynie w odniesieniu do lokalnych plików.

Zapiszmy na dysku przykładowy plik znajdujący się pod adresem: <http://www.enwo.pl/przetwarzanie/dane> a następnie wczytajmy go do obiektu `poznan`

```
poznan <- readRDS("dane/pm10_new.rds")
head(poznan,4) # pokazuje tylko 4 pierwsze linie ramki danych
```

##	data	yy	mm	dd	tmax	tmin	tavg	tdavg	rhavg	wd	ws	gust	slp
## 1	2005-01-01	2005	1	1	5.9	3.6	4.9	4.3	95.7	WSW	16.8	NA	1019.3
## 2	2005-01-02	2005	1	2	6.1	0.1	3.6	1.8	89.7	WSW	13.3	NA	1016.7
## 3	2005-01-03	2005	1	3	6.8	1.1	4.1	-0.2	74.4	W	32.0	68.4	1011.2
## 4	2005-01-04	2005	1	4	6.2	2.0	3.8	1.5	84.7	W	28.4	NA	1015.8
##	prec	totcl	lowcl	sun	vis	snw	ogrodypm10	polankapm10	polankapm25				
## 1	3.5	8.0	8.0	0.0	4.6	0	25.61364	21.543478				NA	
## 2	0.4	7.0	7.0	0.0	9.8	0	19.56250	12.291667				NA	


```
## 3 0.6 5.5 5.5 1.8 15.9 0 10.91667 9.666667 NA
## 4 4.7 7.2 7.2 1.2 13.8 0 12.73913 8.395833 NA
##   poznanpm10 maxwind wd_deg ws_avg   u     v przekroczenie50poz pm10_m1
## 1      25.61    16.8 275.59  3.83 3.81 -0.37      0    16.43
## 2      19.56    13.3 253.07  7.58 7.25  2.21      0    25.61
## 3      10.92    68.4 267.56  7.83 7.82  0.33      0    19.56
## 4      12.74    28.4 271.87  9.04 9.04 -0.29      0    10.92
##   pm10_m2 pm10_m3 pm10_m4 pm10_m5 pm10_m6 pm10_m7
## 1    10.60    7.56    11.16    19.44    17.35    12.38
## 2    16.43    10.60    7.56    11.16    19.44    17.35
## 3    25.61    16.43    10.60    7.56    11.16    19.44
## 4    19.56    25.61    16.43    10.60    7.56    11.16
```

1. Narysuj histogram koncentracji dobowej PM10 w Poznaniu.
2. Oblicz wartości statystyk testowych dla wczytanego zbioru danych.

5.3 Zapis danych

Wspomniane powyżej instrukcje wczytywania danych mają najczęściej swoje odpowiedniki dla zapisu poprzez zamianę słowa kluczowego funkcji `read` lub `load` na `write` lub `save`. Jako że zapis danych zwykle jest łatwiejszy niż wczyt zagadnienie to zostanie omówione w dużym skrócie w kolejnych częściach niniejszego rozdziału. W przypadkach wątpliwych zawsze warto korzystać z wbudowanego systemu pomocy lub z podręcznika P. Biecka (Biecek, 2016).

5.3.1 Dane tekstowe (csv i pokrewne)

Zapis zmiennej będącej wektorem, macierzą lub ramką danych do postaci tekstowej możliwy jest za pomocą funkcji `write.table()` lub `write.csv()` / `write.csv2()`. Argumenty funkcji dopasowujące format zapisu i ich wartości domyślne można sprawdzić za pomocą polecenia `?write.table`. W zdecydowanej większości są one analogiczne lub bardzo podobne do argumentów funkcji wczytujących.

Zadanie kontrolne:

1. Wczytaj do **R** zbiór danych ze średnimi temperaturami powietrza w Polsce w latach 1971-2000 (np. z pliku `p11.csv`) i nazwij go `dane`
2. Zaokrągł wartości temperatur powietrza do 1 miejsca po przecinku i zapisz je jako obiekt `dane2`
3. Wyświetl pierwsze kilka wierszy nowej ramki danych funkcją `head()` w celu upewnienia się co do poprawności poprzedniej operacji
4. Zapisz obiekt `dane2` do pliku `dane2.txt`. Kolumny rozdziel znakiem `;`, miejsca dziesiętne przecinkiem. W pliku będą zawarte nazwy kolumn, bez nazwy wierszy.

Przykładowe rozwiązanie:

```
dane <- read.csv("p11.csv")
dane2 <- round(dane, 1)
head(dane2)
```

```
##   rok   I   II  III  IV   V   VI  VII VIII  IX  X  XI  XII
## 1 1971 -3.0  0.4 -0.1 7.4 14.5 14.9 17.8 18.7 11.2 8.4 2.6  3.0
## 2 1972 -5.8  0.3  3.9 7.4 12.5 16.2 19.4 16.6 11.4 6.1 4.4  0.0
## 3 1973 -1.7  1.3  3.7 6.1 12.3 15.8 17.6 17.1 13.1 6.6 1.8 -0.7
## 4 1974  0.1  2.3  4.4 6.9 10.8 14.0 15.6 17.7 13.4 6.3 3.9  2.7
## 5 1975  3.0 -0.6  3.9 6.6 13.3 15.4 18.6 18.3 15.8 8.0 1.8  0.9
## 6 1976 -2.0 -3.3 -0.8 6.7 11.9 15.1 18.1 15.3 12.6 7.6 4.7 -1.2
```

```
write.table(dane2, file="dane2.txt", sep = ";", dec=",", col.names = TRUE, row.names =
```

5.3.2 Arkusze kalkulacyjne

Format Excela (.xls/.xlsx) jest obsługiwany przez wiele wcześniej wspomnianych pakietów, stąd w zależności od wybranego rozwiązania różne funkcje mogą być użyte do zapisu danych w tym formacie. Poniżej zostanie zademonstrowany przykład zapisu istniejącego obiektu `dane2` przy użyciu biblioteki `xlsx` i funkcji `write.xlsx()`. W najprostszej postaci wymagane jest zadeklarowanie jedynie nazwy obiektu do zapisania oraz nazwy pliku

```
library(xlsx) # aktywowanie pakietu
```

```
## Loading required package: rJava
```

```
## Loading required package: xlsxjars  
write.xlsx(dane2, file = "nowyplik.xlsx")
```

Należy przy tym pamiętać, że praca na plikach .xls/.xlsx umożliwia jedynie zapis/odczyt danych. Grafika (np. wykresy) jest bezpowrotnie tracona, a w niektórych przypadkach może powodować także dodatkowe problemy z pracą na pliku.

5.3.3 Formaty natywne R

5.3.3.1 .Rdata

Domyślnie pliki w programie **R** są zapisywane z rozszerzeniem **.Rdata** lub **.rda**. Funkcja zapisująca dane w tym formacie to **save()**, która wymaga zadeklarowania przynajmniej 2 elementów: nazwy zapisywanego obiektu/zmiennnej oraz nazwy pliku docelowego.

Jeśli chcemy analogicznie jak w podpunkcie 5.3.2 zapisać wyniki przechowywane w zbiorze **dane2** instrukcja powinna wyglądać następująco:

```
save(dane2, file="dane2_test.Rdata")
```

Dużą zaletą formatu natywnego **R** jest możliwość zapisania więcej niż 1 obiektu do pliku. Listę obiektów do zapisania należy zadeklarować kolejno po przecinku, lub w formie listy plików. Spróbujmy oprócz zbioru **dane2** zapisać wektor małych liter przechowywanych w zmiennej **letters**:

```
save(letters, dane2, file="dane2_i_litery.Rdata")
```

Jeśli chcemy zapisać wszystkie obiekty jakie mamy widoczne załadowane w **R** (tj. widoczne w górnym prawym rogu w oknie **Environment**) zamiast wpisywać jeden po drugim do funkcji **save()**, możemy wykorzystać funkcję **save.image()**, która domyślnie podstawia nazwy wszystkich obiektów do instrukcji **save()**.

Jest to w zasadzie bardzo podobna opcja do wyskakującego okna przy zamykaniu programu **RStudio**, które pyta czy chcemy zapisać nasze zbiory danych widoczne w **Environment**. Jeśli wybierzemy opcję potwierdzającą, wówczas przy ponownym uruchomieniu programu **RStudio** zostaną automatycznie wczytane

wszystkie poprzednie zbiory danych. Domyślnie dane te są zapisywane w naszym bieżącym katalogu roboczym w ukrytym pliku `.RData`.

5.3.3.2 RDS

Wygodnym rozwiązaniem pozwalającym na zapis pojedynczego obiektu w formacie `.rds` jest funkcja `saveRDS()`. Poziom kompresji i szybkości jest podobny jak przy plikach `.Rdata`, a jednocześnie możemy sami decydować o nazwie wczytywanego obiektu. Poniżej zaprezentowano przykładowe zastosowanie dla zapisu zbioru `dane2` w formacie `.rds`

```
saveRDS(dane2, file="dane2_test.rds")
```

Zadanie domowe

Zapoznaj się z rozdziałem 2.6 (strony 50-67) z podręcznika P. Biecka (Biecek, 2016). Podręcznik dostępny online na stronie <http://www.biecek.pl/R/PrzewodnikPoPakiecieRWydanieIVinternet.pdf>

Zadanie sprawdzające

<https://goo.gl/forms/PDlghTsu8MoPCrjC3>

Rozdział 6

Pętle

Niewątpliwą przewagą **R** nad typową pracą w arkuszach kalkulacyjnych (lub innych “klikanych” programach) jest możliwość automatyzacji przetwarzania danych poprzez wykorzystanie pętli programistycznych.

Często zdarza się, że chcemy wykonać jakąś operację więcej niż raz, zmieniając tylko fragmentarycznie zakres naszego postępowania. Przykładowo, możemy chcieć wykonać oddzielne wykresy dla każdej kolumny, rzędu, policzyć ustalone statystyki dla wszystkich plików znajdujących się w naszym katalogu roboczym, itp.

Rozwiązaniem w takich przypadkach jest pętla programistyczna pozwalająca na cykliczne uruchamianie zdefiniowanego ciągu instrukcji, np. określoną liczbę razy lub do momentu wystąpienia pewnych warunków. W ten sposób komputer będzie mógł wykorzystać swoją najmocniejszą stronę - wielokrotnie powtarzać przekazane zadanie (Biecek, 2016), które człowiekowi mogłoby zająć kilka żyć.

W **R**, jak w każdym dojrzałym języku programowania, dostępnych jest kilka rodzajów pętli programistycznych: **repeat**, **while** oraz **for** (Gągolewski, 2016). Ze względu na specyfikę naszego kursu skoncentrujemy się na tych dwóch ostatnich, które są najczęściej stosowane w praktyce.

6.1 Pętla *for*

Najpopularniejsza pętla w językach programowania to tzw. pętla **for**. Pozwala ona na podstawienie za każdym razem dowolnie zdefiniowanych wartości do zmiennej podczas uruchamiania określonego bloku kodu. Choć brzmi to dość skomplikowanie poniższy przykład pokaże istotę zagadnienia.

Weźmy najprostszą funkcję, np. `print()`, której zadaniem jest wyświetlenie wartości argumentu umieszczonego w nawiasie. Jeśli chcemy wyświetlić za pomocą tej funkcji (w kolejnych liniach) liczby od 1 do 4 to musielibyśmy wpisać 4 komendy:

```
print(1)
print(2)
print(3)
print(4)
```

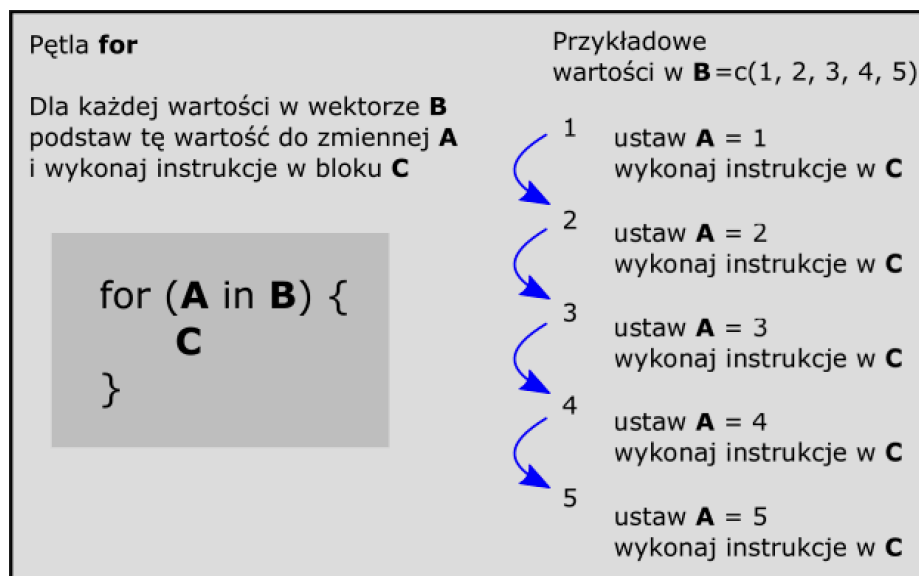
Powyższy sposób działa, choć łatwiej można do tego celu wykorzystać pętlę **for**, za każdym razem podstawiając wartości od 1 do 4. Ten sam efekt z pętlą **for** wyglądałby następująco:

```
for(zmienna in 1:4) {
    print(zmienna)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

Wyjaśnienie: Po inicjacji pętli za pomocą słowa kluczowego **for** w nawiasie okrągłym należy wpisać nazwę nowej zmiennej (tu: **zmienna**). Następnie umieszcza się słowo kluczowe **in**, po którym należy wskazać lub zdefiniować jakie wartości będzie przyjmować zmienna (tu: **1:4**).

Polecenia znajdujące się pomiędzy znakami `{ }` są wykonywane dla każdej zdefiniowanej wartości zmiennej, taką ilość razy jaka jest liczba elementów wektora znajdującego się na prawo od słowa kluczowego **in**. Szczegółowy schemat działania pętli **for** przedstawiono na rys. 6.1.



Ciekawostka: Teoretycznie kod instrukcji pętli można umieścić w jednym ciągu, w tej samej linii co komendę `for`, np.:

```
for(zmienna in 1:4) print(zmienna)
```

Jednak bardziej czytelny zapis z blokiem pomiędzy nawiasami klamrowymi sprawia, że schemat jednolinijkowy jest stosunkowo rzadko stosowany.

6.1.1 Zadania treningowe - część I

1. Napisz pętlę, która wyświetli napis *R jest (tu wstaw dowolne słowo...)* 100 razy
 2. Napisz pętlę, która za pomocą funkcji `sample()` “puści kupon totolotka” 10 razy. Jeśli brakuje wyniku postaraj się wymusić wyświetlenie stworzonej instrukcji za pomocą funkcji `print()`
 3. Napisz pętlę wyświetlającą zmienną *i* o wartościach 3.5, 5 i 20. Pamiętaj, że wartości definiowane po słowie kluczowym `in` również są wektorem!
-

6.1.2 Zadania treningowe - część II

Pętla `for` może być wykorzystana nie tylko w przykładach teoretycznych, ale także w praktyce... Wyobraźmy sobie, że chcemy wykonać wykres liniowy średniej miesięcznej temperatury powietrza w Polsce dla określonego miesiąca w latach 1971-2000.

Wykonaj poniższe kroki:

1. Wczytaj do obiektu `dane` zbiór danych znajdujący się pod adresem: `http://enwo.pl/przetwarzanie/dane/pl1.csv`
2. Wykres w najprostszej formie w **R** można wykonać za pomocą funkcji `plot()`, gdzie jako argumenty parametrów `x` oraz `y` należy podać wektory wartości położenia punktów na osiach `x` i `y`. Dodatkowo warto wykorzystać argument `type='b'`, który połączy punkty (domyślny rodzaj wykresu) liniami. Wykonaj wykres dla stycznia. W razie konieczności skorzystaj z systemu pomocy dla funkcji `plot()`
3. Powtórz krok 2. wykonując wykres dla lutego. Dopisz dla funkcji `plot()` argument `main="2"`
4. Korzystając z systemu pomocy zoptymalizuj wykres tak, aby zawierał poprawne podpisy osi `x` i `y`
5. ... **Za pomocą pętli `for` napisz kod, który stworzy wykresy dla wszystkich 12-tu miesięcy**

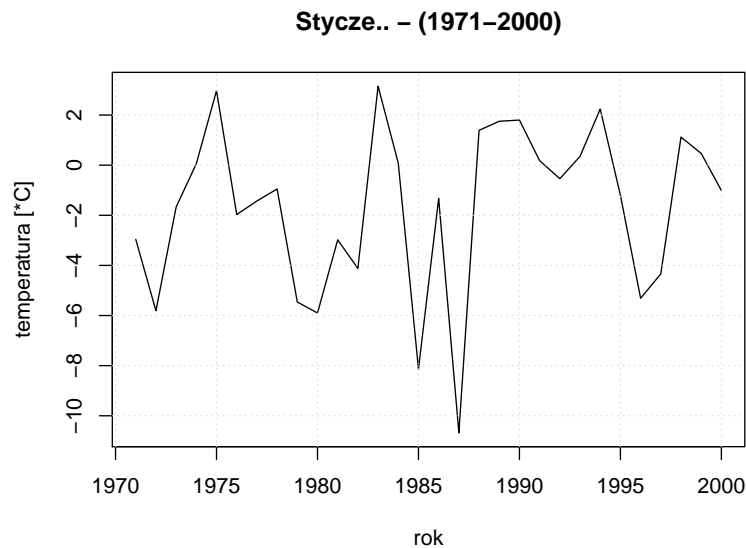
```
## Warning in title(...): conversion failure on 'Styczeń - (1971-2000)' in
## 'mbcsToSbcs': dot substituted for <c5>
```

```
## Warning in title(...): conversion failure on 'Styczeń - (1971-2000)' in
## 'mbcsToSbcs': dot substituted for <84>
```

6. Jak w punkcie 5., ale zamiast wykresów stwórz histogramy oraz wyświetl w konsoli wynik dla średniej miesięcznej temperatury powietrza

6.1.3 Pętla w pętli

Często spotykanym zwyczajem (choć z początku mało intuicyjnym) jest zagnieżdżanie pętli. Dla czytelności kodu niektóre programy (jak np. **RStudio**) automatycznie stosują tzw. wcięcia



Rysunek 6.1: Temperatura stycznia w Polsce, 1971-2000

Sprawdź działanie poniższego kodu, który wygeneruje wszelkie możliwe kombinacje nazw stacji, lat i miesięcy w latach 2016-2017:

```
for(stacja in c("Poznań", "Łeba")){
  for(rok in c(2016,2017)){
    for(miesiac in 1:12){

      calosc <- paste(stacja, rok, miesiac) # funkcja paste "złącza" ciagi tekstowe
      print(calosc)

    } # ten nawias domyka petle dla zmiennej `miesiac`
  } # ten nawias domyka petle dla zmiennej `rok`
} # ten nawias domyka petle dla zmiennej `stacja`
```

6.1.4 Wykorzystanie pętli for do pobierania danych

Wiele danych meteorologicznych można pobrać za darmo z internetu. Dane te są często w bardzo różnych formatach danych, dlatego najczęściej aby móc na nich pracować należy je pobrać na dysk.

Jednym z najczęściej pobieranych zbiorów danych są reanalizy meteorologiczne NCEP/NCAR (Kalnay et al., 1996) (<https://scholar.google.pl/citations?user=hLLKbYIAAAAJ&hl=en&oi=sra>). Dane te można pobrać jako oddzielne pliki dla poszczególnych elementów ze strony: <ftp://ftp.cdc.noaa.gov/Datasets/ncep.reanalysis.dailyavgs/surface/>

Każdy z plików z danymi jest zapisany jako **NAZWA_PARAMETRU.ROK.nc**. Znając ten schemat nazywania plików możemy pobrać dane np. dla temperatury powietrza (*air.sig*) kopiując do schowka cały adres do pierwszego pliku (<ftp://ftp.cdc.noaa.gov/Datasets/ncep.reanalysis.dailyavgs/surface/air.sig995.1948.nc>) i próbując za każdym razem podstawić w tym adresie zamiast liczby 1948 lata (liczby) które chcemy pobrać. Pobranie pliku w **R** umożliwia funkcja `download.file()`. Z kolei do łączenia nazw użyjemy funkcji `paste()`.

1) spróbujemy złączyć ciąg tekstowy dla pobieranego adresu:

pierwsza część adresu:

```
adres1 <- "ftp://ftp.cdc.noaa.gov/Datasets/ncep.reanalysis.dailyavgs/surface/air.sig995."
```

```
rok <- 2016 # jakis rok, który chcemy pobrać
```

```
adres2 <- ".nc" # dołączanie końcówki adresu
```

```
link <- paste(adres1,rok,adres2) # łączymy adres do finalnej postaci:
```

```
link
```

```
## [1] "ftp://ftp.cdc.noaa.gov/Datasets/ncep.reanalysis.dailyavgs/surface/air.sig995.2016.nc"
```

Ooopss...

Okazuje się, że domyślnie funkcja paste() stosuje spacje pomiędzy

złączanymi ciągami tekstu. Można to zmienić za pomocą opcji 'sep'

```
link <- paste(adres1,rok,adres2, sep="") # poprawiamy link aby nie miał spacji
```

```
link
```

```
## [1] "ftp://ftp.cdc.noaa.gov/Datasets/ncep.reanalysis.dailyavgs/surface/air.sig995.2016.nc"
```

2) Skoro nazwa jest poprawna spróbujemy użyć funkcji download.file()

do pobrania wygenerowanego linku. Musimy dodatkowo wygenerować nazwę

pliku, który będzie zapisany na dysku

```
plik <- paste(rok,adres2,sep="")
```

Zadanie treningowe/domowe:

1. Korzystając z pętli `for` pobierz pliki dla temperatury powietrza z reanaliz NCEP/NCAR dla lat 2016-2010
2. W linku: enwo.pl/przetwarzanie/dane/pl2013.zip zapisano dane meteorologiczne dla głównych polskich stacji meteorologicznych z roku 2013. Rozpakuj te dane na dysku a następnie napisz pętlę programistyczną która:
2.1. Wczyta te pliki do **R** (przydatne może być przekazanie wyniku funkcji `dir()` do zmiennej) 2.2. Wyświetli średnią arytmetyczną z wartości zapisanych w drugiej kolumnie w każdym z plików

6.2 Pętla *while*

Kolejną pętlą, nieco rzadziej stosowaną, jest pętla `while`. Wykonuje się ona tak długo, aż testowany warunek logiczny przestanie zwracać wartość `TRUE`. W niektórych przypadkach takie rozwiązanie może powodować potencjalne zagrożenie, gdyż pętla może działać w nieskończoność.

Ogólna postać pętli `while` wygląda bardzo podobnie do pętli `for`, z tą różnicą że w nawiasie testowany jest warunek logiczny, a w przypadku testowania warunku logicznego na zmiennych muszą one być zadeklarowane przed instrukcją `while`.

Sprawdźmy to na przykładzie:

```
zmienna <- 1
while(zmienna>0){          # sprawdz warunek logiczny, czy zmienna jest > 0
                           # jesli tak, to uruchom to co jest pomiedzy {}

    print(zmienna)         # wyswietlam aktualna wartosc zmiennej
    zmienna <- zmienna-0.3 # zmniejszam wartosc zmiennej o 0.3
}
```

```
## [1] 1
## [1] 0.7
## [1] 0.4
```

```
## [1] 0.1
```

Pętla została uruchomiona 4 razy ponieważ za pierwszym razem przy testowaniu warunku logicznego zmienna miała wartość 1 i zmniejszała się aż do 0.1, po czym przy ostatnim uruchomieniu pętli zmienna otrzymała wartość -0.2 i warunek logiczny `zmienna>0` zapisany w instrukcji `while` dał wartość `FALSE` i pętla zaprzestała swojego działania.

6.2.1 Przykład z pętlą *while*

Zagrajmy z komputerem w totolotka.

1. Wybierz wektor 6 unikalnych wartości w przedziale od 1 do 49. Zapisz wynik do obiektu `kupon`
2. Wylosuj 6 liczb z rozkładu jednostajnego płaskiego za pomocą funkcji `sample()` i zapisz wynik do obiektu `wynik`
3. Sprawdź czy wygrałeś poprzez działanie w krokach 4-6:
4. Dokonaj konkatencji obiektów `wynik` i `kupon` do obiektu `razem`
5. Sprawdź za pomocą funkcji `duplicated()` czy obiekt `razem` zawiera `>2` powtórzenia. Wynik funkcji `duplicated()` daje wartości logiczne, które można zsumować (`TRUE = 1`, `FALSE = 0`)
6. Zapisz wynik otrzymany w poprzednim punkcie do zmiennej `wygrana`
7. Całe postępowanie wprowadź do pętli `while`, która będzie się uruchamiać tak długo, aż nie wygrasz (tzn. trafisz choć 3 liczby)
8. ... Policz czy kto wygrał ...

Przykładowe rozwiązanie:

```
nr_zakladu <- 0 # sprawdzmy ile totolotkow musimy puscic zanim wygramy chocby 'trojke'
kupon <- c(10, 17, 23, 26, 39, 49)
wygrana <- 0
while(wygrana<3){ # jesli bedzie >= 3 to znaczy ze wygralismy,
                  # w przeciwnym razie gramy jeszcze raz
  wynik <- sample(1:49, 6)
  razem <- c(wynik, kupon)
  wygrana <- sum(duplicated(razem))
}
```

```
nr_zakladu <- nr_zakladu +1

cat(c(nr_zakladu,"wylosowane liczby to:", wynik,"\n"))
}
```

```
## 1 wylosowane liczby to: 14 18 27 42 10 40
## 2 wylosowane liczby to: 47 32 30 3 10 8
## 3 wylosowane liczby to: 34 19 37 23 33 44
## 4 wylosowane liczby to: 19 38 44 10 30 6
## 5 wylosowane liczby to: 14 19 1 18 40 15
## 6 wylosowane liczby to: 24 29 49 9 38 30
## 7 wylosowane liczby to: 39 6 35 19 37 29
## 8 wylosowane liczby to: 39 27 25 37 2 21
## 9 wylosowane liczby to: 36 34 23 40 20 11
## 10 wylosowane liczby to: 4 5 15 24 30 18
## 11 wylosowane liczby to: 45 15 22 16 30 12
## 12 wylosowane liczby to: 24 37 4 41 16 48
## 13 wylosowane liczby to: 17 49 23 42 39 18
```

Zadanie domowe

Pętla while - pobierz dane IMGW-PIB

Na stronie <https://dane.imgw.pl/> istnieje możliwość pobrania danych meteorologicznych ze zbiorów IMGW-PIB. Jednorazowo można pobrać z poziomu przeglądarki tylko fragment całości udostępnionych danych. Pobieranie danych dla dłuższych okresów czasu wymagałoby żmudnej i czasochłonnej pracy, stąd też można do tego celu wykorzystać dowolną pętlę programistyczną i funkcję `download.file()`. Dla ułatwienia działania kod automatyzujący pobieranie danych zapisano w postaci pakietu `imgw` dostępnego na stronie <http://github.com/bczernecki/imgw>. W celu sprawdzenia działania niniejszego rozwiązania wykonaj poniższe polecenia:

1. Zainstaluj i następnie aktywuj pakiet `devtools`. Pozwola on na instalowanie także nieoficjalnych paczek R.

2. Za pomocą instrukcji `install_github("bczernecki/imgw")` zainstaluj pakiet nazwany `imgw` i następnie go aktywuj.
3. Zapoznaj się z dokumentacją do tego pakietu i pobierz przykładowe dane meteorologiczne.
4. Przeanalizuj kod wpisując w konsoli samą nazwę funkcji pobierającej (tj. `pobierz` - bez nawiasów i argumentów). Sprawdź sposób działania funkcji który wyświetli się w konsoli. Utwórz nowy skrypt i sprawdzaj działanie kodu funkcji `pobierz` uruchamiając poszczególne instrukcje linia po linii. Unikaj pętli w celu lepszego zrozumienia sposobu działania kodu.

Zadanie sprawdzające

Rozdział 7

Czas i napisy

W meteorologii i klimatologii niezwykle ważne jest analizowanie zbiorów danych w aspekcie czasowym. Przed rozpoczęciem analizy szeregów czasowych niezbędne jest zrozumienie sposobu przechowywania tego typu danych przez komputer.

W **R** powszechnie stosowane są 2-3 formaty przechowywania czasu:

- Klasa **Date** - wystarczająca jeśli nie mamy danych o rozdzielczości mniejszej niż dobową (*sub-daily*) oraz nasze dane nie wymagają różnic w zmianach czasu urzędowego
- Klasy POSIX-owe: **POSIXct** oraz **POSIXlt** (odpowiednio: *calendar* i *local time*). Zdecydowanie częściej stosowana jest klasa **POSIXct**, która pozwala na przechowywanie czasu z dokładnością do sekundy (lub jej części ułamkowych). Poza tym obsługuje kwestie problematyczne związane z różnymi strefami czasowymi oraz zmianą czasu.

7.1 Date

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"
```

Jeśli chcemy zmienić sposób deklarowania daty możemy użyć argumentu `format`

```
as.Date("02/22/2015", format = "%m/%d/%Y")
```

```
## [1] "2015-02-22"
```

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()`.

Zadanie sprawdzające

1. Utwórz dowolnie nazwany obiekt przechowujący dzisiejszą datę
2. Dodaj lub odejmij od niego dowolną liczbę całkowitą oraz wektor w postaci sekwencji liczb wygenerowanych za pomocą operatora `:`
3. W **R** daty przechowywane są również jako liczby. Sprawdź działanie funkcji `as.numeric()` na dowolnym obiekcie klasy `Date`. Jaki dzień to dla komputera 0?

7.1.1 W jaki sposób najszybciej utworzyć ciąg dat?

Bardzo często zdarza się, że konieczne jest wygenerowanie ciągu dat, które będziemy umieszczać w jednej z kolumn ramki danych. Podobnie jak w przypadku “normalnych” wektorów również do tego celu możemy zastosować funkcję `seq`, która w przypadku operacji na datach ma także postać rozszerzoną jako `seq.Date()`. Sprawdź pomoc dla tej funkcji. Następnie:

1. Utwórz datę początkową i końcową dla generowanego ciągu dat:

```
poczatek <- as.Date("2016-01-01")  
koniec <- as.Date("2016-12-31")
```


2. Podłącz wygenerowane obiekty jako argumenty `from` oraz `to` dla funkcji `seq.Date()` oraz opcję `by` jako dzień (ang. *day*):

```
naszedaty <- seq.Date(from=początek, to=koniec, by="day")
head(naszedaty)
```

```
## [1] "2016-01-01" "2016-01-02" "2016-01-03" "2016-01-04" "2016-01-05"
## [6] "2016-01-06"
```

3. Czasem jednak niezbędne jest wygenerowanie dat w pewnych interwałach czasu, np. co tydzień, albo co kilka- kilkanaście dni. Do tego celu można użyć opcji `by`. Przykładowo, daty z interwałem co tydzień, można wygenerować wg jednej z poniższych opcji:

```
cotydzien1 <- seq.Date(from=początek, to=koniec, by="week")
cotydzien2 <- seq.Date(from=początek, to=koniec, by="7 days")
head(cotydzien1)
```

```
## [1] "2016-01-01" "2016-01-08" "2016-01-15" "2016-01-22" "2016-01-29"
## [6] "2016-02-05"
```

```
head(cotydzien2)
```

```
## [1] "2016-01-01" "2016-01-08" "2016-01-15" "2016-01-22" "2016-01-29"
## [6] "2016-02-05"
```

7.1.2 Operowanie datami

Wiele operacji na datach nie należy do trywialnych lub intuicyjnych. Przykładowo, jeśli chcemy *wyciągnąć* z obiektu klasy `Date` jaki to jest dzień tygodnia możemy posłużyć się opcją formatowania opisaną w systemie pomocy pod hasłem `?strptime`. Sprawdźmy na przykładzie dla obiektu `naszedaty`:

```
dnitygodnia <- format(naszedaty, "%A")
head(dnitygodnia)
```

```
## [1] "Friday"    "Saturday"  "Sunday"    "Monday"    "Tuesday"   "Wednesday"
```

7.1.3 Biblioteka lubridate

Znaczna część operacji na datach jest usprawniona dzięki pakietom **R**, dedykowanym do pracy na obiektach zawierających czas. Jednym z nich jest biblioteka lubridate. Aby z niej korzystać należy ją zainstalować i aktywować.

```
#install.packages("lubridate") # jeśli nie był wcześniej zainstalowany  
library(lubridate)
```

```
##  
## Attaching package: 'lubridate'  
  
## The following object is masked from 'package:base':  
##  
##      date
```

Przyjrzyjmy się działaniu przykładowych funkcji z tego pakietu:

```
now()  
  
## [1] "2018-06-21 21:42:51 CEST"
```

```
today()
```

```
## [1] "2018-06-21"
```

```
dzis <- today()
```

```
week(dzis)
```

```
## [1] 25
```

```
day(dzis)
```

```
## [1] 21
```

```
month(dzis)
```

```
## [1] 6
```

```
months(dzis)
```

```
## [1] "June"
```

```
year(dzis)
```

```
## [1] 2018
```

Zadanie

1. Oblicz ile dni trwała druga wojna światowa w Europie (1. września 1945 - 8 maja 1945).
2. Sprawdź jaki dzień tygodnia będzie za 100 dni od dziś

7.2 POSIXct

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")
```

```
czas1
```

```
## [1] "2015-02-13 12:56:26 CET"
```

```
czas2 <- as.POSIXct("14022015 12:56:26", format = "%d%m%Y %H:%M:%S")
```

```
czas2
```

```
## [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).

```
czas2 - czas1
```

```
## Time difference of 1 days
```

Możemy także sprawdzić lub nadać właściwości związane ze strefą czasową:

```
tz(czas2)
```

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

... choć niektóre rzeczy łatwiej zrobić za pomocą klasycznych instrukcji POSIXowych. Sprawdźmy różnicę czasu pomiędzy Jerozolimą i Warszawą w dn. 21 czerwca:

```
warszawa <- as.POSIXct("2017-06-21 16:00:00", tz = "Europe/Warsaw")
jerezolima <- as.POSIXct("2017-06-21 16:00:00", tz = "Asia/Jerusalem")
jerezolima-warszawa
```

```
## Time difference of -1 hours
```

Zadanie domowe

1. Wczytaj plik z danymi dostępnymi na stronie <http://enwo.pl/przetwarzanie/dane/poz.txt>.
2. Następnie przekonwertuj kolumnę/kolumny reprezentujące czas na nową kolumnę zawierającą czas jako obiekt klasy `Date`. Możesz także utworzyć więcej nowych kolumn w zależności od własnych potrzeb.
3. Oblicz średnią wartość temperatury dla wybranego (jednego miesiąca).

Zadanie podsumowujące

TBA

Rozdział 8

dplyr - część I

Wiele operacji wykonywanych na ramkach danych ma bardzo podobną strukturę. Bardzo często zbiory danych łączymy, filtrujemy, rozdzielamy, grupujemy, obliczamy wybrane statystyki i wizualizujemy w poszukiwaniu istoty problemu. Wiele z tych operacji wykonywanych na ramkach danych można wykorzystać za pomocą natywnie dostępnych funkcji **R**. Można także skorzystać z kilku-, kilkunastu pakietów bardzo popularnych wśród analityków danych, które skracają czas obliczeń o 80% i pozwalają na zawarcie istoty problemu w zaledwie kilku liniach kodu...

Jedną z najczęściej wykorzystywanych bibliotek do analizy danych jest biblioteka **dplyr** (Wickham and Francois, 2016), która w dużym stopniu zrewolucjonizowała analizę danych w środowisku **R**. Poniżej zostanie przedstawionych kilka praktycznych przykładów związanych z wykorzystaniem wybranych funkcji z tego pakietu.

*Pakiet **dplyr** nie jest dostępny w domyślnym środowisku **R**, stąd też wymagana jest jego wcześniejsza, jednorazowa instalacja oraz aktywacja.*

8.1 Łączenie ramek danych - `left_join()`

Do połączenia dwóch ramek danych na podstawie wspólnego identyfikatora można użyć natywnie dostępnej w **R** funkcji `merge()` lub skorzystać z jednej z od-

mian funkcji `join` dostępnej w bibliotece `dplyr`. Najczęściej stosowana komenda to `left_join()`, która zwraca wszystkie elementy z pierwszej ramki danych oraz wszystkie kolumny z obu łączonych ramek danych.

Spójrzmy na schemat działania tej funkcji na podstawie poniższego przykładu z dwiema ramkami danych zawierających

```
df1 <- data.frame(id_stacji = c(1:4,3), pomiar1 = c(2.5,1.25,2,3,2))
df2 <- data.frame(id_stacji = c(1, 3, 5), pomiar2 = c(10.2,9.6, 12.3), inne = letters[
print(df1)
```

```
##   id_stacji pomiar1
## 1         1     2.50
## 2         2     1.25
## 3         3     2.00
## 4         4     3.00
## 5         3     2.00
```

```
print(df2)
```

```
##   id_stacji pomiar2 inne
## 1         1    10.2    a
## 2         3     9.6    b
## 3         5    12.3    c
```

Jeśli chcemy złączyć obie ramki danych: `df1` oraz `df2` na podstawie kolumny `id_stacji` najszybciej wynik działania dostaniemy działając funkcją `left_join()`:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:lubridate':
##
##   intersect, setdiff, union

## The following objects are masked from 'package:stats':
##
##   filter, lag
```

```
## The following objects are masked from 'package:base':
##
## intersect, setdiff, setequal, union
left_join(df1,df2)
```

```
## Joining, by = "id_stacji"

##   id_stacji pomiar1 pomiar2 inne
## 1         1    2.50    10.2   a
## 2         2    1.25     NA <NA>
## 3         3    2.00     9.6   b
## 4         4    3.00     NA <NA>
## 5         3    2.00     9.6   b
```

Jak widać na powyższym przykładzie funkcja domyślnie poszukała kolumny w obu zbiorach danych o takiej samej nazwie i na tej podstawie dokonała złączenia. Jednocześnie identyfikatory stacji, które nie istniały w drugiej ramce danych zostały uzupełnione jako braki danych (NA). *Jeśli chcielibyśmy uzyskać inne możliwe kombinacje możemy zastosować inne warianty rodziny funkcji join opisanej w systemie pomocy R*

Brak wspólnej nazwy kolumny łączącej

Funkcja `left_join()` domyślnie szuka wspólnych nazw kolumn na podstawie których łączy dwie ramki danych. Jeśli nazwy kolumn po których chcemy dokonać złączenia są różne możemy:

- odpowiednio wcześniej zunifikować te nazwy (np. za pomocą `colnames()`)
- lub wskazać funkcji `left_join()` argument `by=` z kolumnami po których chcemy łączyć

W celu sprawdzenia takiego schematu postępowania pobierz dane z adresu http://enwo.pl/przetwarzanie/dane/przyklad1_join.Rdata i załaduj do środowiska R. W zakładce *Environment* powinny pojawić się 2 nowe obiekty:

- `xym` - zawiera współrzędne geograficzne, wysokości stacji, międzynarodowe kody stacji i nazwy stacji meteorologicznych

- **wynik**- zawiera podsumowanie dobowe dla temperatury maksymalnej, minimalnej i średniej wg depesz SYNOP z godz. 6:00 UTC

```
load("/home/bartosz/github/przetwarzanie/dane/przyklad1_join.Rdata")
head(xym)
```

```
##      lon      lat alt  code      name
## 1 27.95000 55.81667 133 26554 Verhnedvinsk
## 2 27.46667 55.36667 131 26643 Sarcovschina
## 3 26.31667 55.05000 209 26645      Lyntupy
## 4 28.76667 55.46667 133 26653      Polock
## 5 27.75000 54.88333 197 26657      Dokshitsy
## 6 28.70000 54.88333 174 26659      Lepel
```

```
head(wynik)
```

```
##      stacja tmax tmin tavg
## 1 Kolobrzeg  6.3  1.4  3.8
## 2 Koszalin   6.6 -2.4  2.1
## 3  Ustka     5.3  0.6  3.6
## 4   Leba     6.1  1.4  3.4
## 5 Darlowek   6.5 -0.4  2.5
## 6  Lebork    6.3 -3.9  1.7
```

Jak widzimy wspólnym polem w obu zbiorach danych są nazwy stacji zawarte w polach: **name** oraz **stacja**. Jeśli potraktujemy zbiór **wynik** jako podstawowy do którego chcemy dołączyć współrzędne geograficzne i kod WMO stacji wówczas samo wpisanie komendy `left_join(xym, wynik)` powinno dać błąd. Konieczne jest wskazanie nazw kolumn w zbiorze pierwszym i odpowiadającej mu nazwy kolumny w zbiorze drugim w dość nieintuicyjnej składni argumentu `by =` :

```
calosc <- left_join(wynik, xym, by = c("stacja" = "name"))
```

```
## Warning: Column `stacja`/`name` joining factors with different levels,
## coercing to character vector
```

```
head(calosc)
```

```
##      stacja tmax tmin tavg      lon      lat alt  code
## 1 Kolobrzeg  6.3  1.4  3.8 15.58333 54.18333   3 12100
```



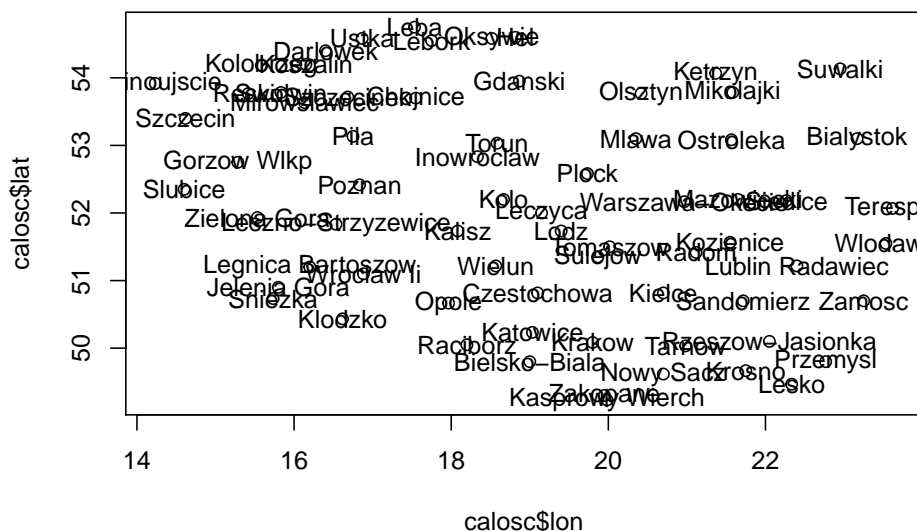
```
## 2 Koszalin 6.6 -2.4 2.1 16.15000 54.20000 32 12105
## 3 Ustka 5.3 0.6 3.6 16.86667 54.58333 6 12115
## 4 Leba 6.1 1.4 3.4 17.53333 54.75000 2 12120
## 5 Darlowek 6.5 -0.4 2.5 16.40000 54.40000 2 12124
## 6 Lebork 6.3 -3.9 1.7 17.75000 54.55000 17 12125
```

Często po złączeniu dwóch ramek danych nasz nowy zbiór zawiera braki. Jeśli chcemy się ich pozbyć możemy użyć funkcji `na.omit()`, która usunie wszystkie rzędy z wartościami NA.

```
calosc <- na.omit(calosc)
```

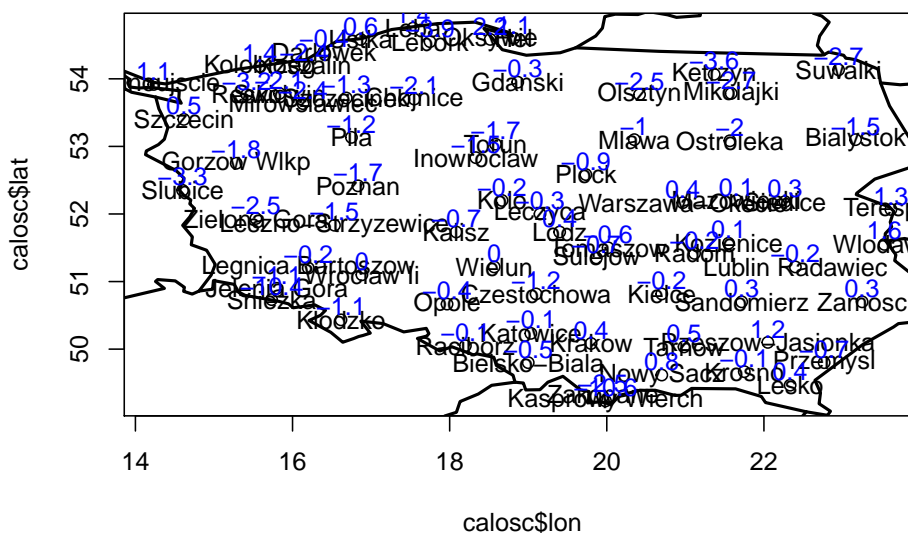
W ten sposób nasza baza danych powinna zawierać tylko poprawne wartości. Możemy w szybki sposób zwizualizować nasz zbiór danych za pomocą wcześniej poznanej funkcji `plot`, gdzie jako współrzędnej `x` i `y` podamy wartości odpowiednio długości (lon) i szerokości geograficznej (lat). Możemy także dodać dowolną informację w postaci tekstowej, np. temperaturę minimalną za pomocą funkcji `text()`. Działa ona analogicznie jak funkcja `plot()` przy czym konieczne jest podanie dodatkowego argumentu `labels=`, który ma być wyświetlony we wskazanych koordynatach. Jeśli chcemy wyświetlić dodatkowo nazwy stacji wówczas możemy zastosować zarówno poniższy kod:

```
plot(x = calosc$lon, y = calosc$lat); text(x = calosc$lon, y = calosc$lat, labels=calosc$stacja)
```



- Można dodatkowo dodać fragment kodu, który doda kontury krajów. Jeśli nie mamy nigdzie w pobliżu odpowiednio przygotowanej warstwy w postaci pliku GISowego, możemy wykorzystać pakiet `mapdata`, w którym znajdują się podstawowe dane z granicami administracyjnymi dla całego świata:

```
#install.packages("mapdata") # jeśli chcemy uzyc po raz pierwszy musimy ja zainstalować
library(mapdata) # aktywacja pakietu
plot(x = calosc$lon, y = calosc$lat)
text(x = calosc$lon, y = calosc$lat, labels=calosc$stacja) # rysujemy to co wcześniej
map("world", add=TRUE, lwd=2) # ważne aby ustawić opcję 'add'; reszta parametrów jak d
text(x = calosc$lon, y = calosc$lat+0.2, labels=calosc$tmin, col="blue") # dodajmy jes
```



8.1.1 Łączenie danych po dacie

W katalogu <http://enwo.pl/przetwarzanie/dane/opady/> znajdują się pliki do dzisiejszego ćwiczenia. Zawierają one dobowe sumy opadów atmosferycznych z kilku wybranych polskich stacji. Każdy z plików ma taką samą strukturę zawierającą w kolejnych kolumnach: numer stacji, nazwę stacji, datę oraz sumę opadu. Wartości są rozdzielone znakami tabulacji, a miejsca dziesiętne są oddzielone kropkami.

```
## "V1" "V2"      "V3"      "V4"
## 249180120 "SKOCZÓW" 19501101 2.1
## 249180120 "SKOCZÓW" 19501105 2.1
## 249180120 "SKOCZÓW" 19501107 2.9
## 249180120 "SKOCZÓW" 19501108 0.5
## 249180120 "SKOCZÓW" 19501111 0.6
## 249180120 "SKOCZÓW" 19501114 12.6
## 249180120 "SKOCZÓW" 19501115 12.3
## 249180120 "SKOCZÓW" 19501116 0.6
## 249180120 "SKOCZÓW" 19501117 0.4
```

Zwróć uwagę, że pliki zawierają informację jedynie o dniach, w których wystąpiły opady na danej stacji (jeśli opadu w danym dniu nie było wówczas jest on pominięty). Naszym celem będzie utworzenie jednolitej, pełnej bazy, ze wszystkimi datami w zakresie występujących dat (bez względu na to czy padało), a wartości opadów dla każdej kolejnej stacji będą umieszczane w kolejnych kolumnach, jak na poniższym schemacie:

```
##          daty brenna chalupki cieszyn
## 1 1951-01-01      NA      12.2      7.8
## 2 1951-01-02      NA      4.8      9.6
## 3 1951-01-03    0.1      0.5      4.3
## 4 1951-01-04    0.5      0.9      7.1
## 5 1951-01-05    5.2      NA      4.0
## 6 1951-01-06   12.9      NA      3.3
```

Dla wielu osób pracujących dotychczas w arkuszach kalkulacyjnych taka postać bazy danych jest najbardziej intuicyjna w obsłudze.

Zanim przystąpisz do tworzenia bazy danych utwórz katalog `opady` (np. na pulpicie) i zapisz do niego pliki znajdujące się pod adresem <http://enwo.pl/przetwarzanie/dane/opady>. Następnie ustaw katalog roboczy **RStudio** aby pliki były dostępne bez konieczności wpisywania pełnej ścieżki.

W poniższej tabeli wypisano nazwy stacji oraz liczbę wierszy w każdym z plików. *Dlaczego nie możemy połączyć plików od razu do postaci macierzy / ramki danych za pomocą komendy `cbind()` / `cbind.data.frame()`?*

```
##      1603 BRENNNA.
```

```
## 1198 CHALUPKI.
## 1198 CHAŁUPKI.
## 2053 CIESZYN.
## 1270 GOCZALKOWICE-ZDROJ.
## 1270 GOCZAŁKOWICE-ZDRÓJ.
## 1921 ISTEbNA-MŁODAGORA.
## 1921 ISTEbNA-MŁODAGÓRA.
## 1947 ISTEbNA-STEĆOWKA.
## 1947 ISTEbNA-STEĆÓWKA.
## 1492 JAWISZOWICE.
## 1717 RUDZICA.
## 1763 SKOCZOW.
## 1763 SKOCZÓW.
## 1493 SZCZYRK.
## 1233 TRZEMESNIA.
## 1233 TRZEMEŚNIA.
## 1856 USTRON-RÓWNICA-WIEŚ.
## 1856 USTRON.
## 2032 WAPIENICA.
## 1237 WARSZOWICE.
## 1817 WISŁA-CENTRUM.
## 1268 WISŁA-GŁĘBCE.
## 1268 WISŁA-GŁĘBCE.
## 695 WISŁAWIELKA.
## 39051
```

Do połączenia 2 ramek danych na podstawie wspólnego identyfikatora można użyć natywnie dostępnej w **R** funkcji `merge()` lub skorzystać z pakietu `plyr`, który oferuje nieco bardziej wydajny algorytm łączenia baz danych. W celu przetestowania jego funkcjonalności niezbędne będzie wykonanie poniższych kroków:

1. Stwórz obiekt `data` z datami od 1. stycznia 1950 r. do 31. grudnia 1960 r.
2. Stwórz ramkę danych `wynik` z jedną kolumną nazwaną `data`, w której będą przechowywane wartości dat (z poprzedniego punktu).
3. Wczytaj pierwszy (dowolny) plik z danymi opadowymi i nazwij go `dane`.

Za pomocą funkcji `colnames()` nazwij w intuicyjny sposób kolumny (np.: “id”, “stacja”, “data”, “opad”).

4. Kolumnę zawierającą datę przekonwertuj do klasy `Date`, aby komputer nie miał problemów ze zrozumieniem, że wartości w tej kolumnie przechowują czas a nie wartości liczbowe.
5. Złącz wynikową ramkę danych z wczytanym plikiem za pomocą funkcji `left_join()` z pakietu `dplyr`.
6. Ponów kroki 3-5 wczytując kolejny plik do istniejącej wynikowej ramki danych

Zadanie domowe

Po opanowaniu złączania ramek danych kontynuuj treść poleceń 1-6 poprzez stworzenie pętli `for`, która będzie wczytywać kolejne pliki z danymi opadowymi oraz dopisywać do wynikowej ramki danych. Finalny wynik zapisz do pliku arkusza kalkulacyjnego z rozszerzeniem `.xls`.

Rozdział 9

dplyr - część II

W tej części zapoznasz się z możliwościami pakietu **dplyr** dowiesz się jak w szybki sposób manipulować ramkami danych.

*Ze względu na częste stosowanie rozwiązań z pakietów **dplyr** oraz **tidyr** najbardziej popularne rozwiązania zawarto w tzw. cheat-sheet'ie* * <https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>*

Dane

Przed przystąpieniem do dalszej części pracy wczytaj do środowiska **R** godzinowe wartości pomiarów dla wybranych stacji meteorologicznych IMGW-PIB (2000-2015). Dane do ćwiczenia przygotowano w formacie RDS pod adresem: <http://enwo.pl/przetwarzanie/dane/synop.rds>. Nazwij wczytywany zbiór jako **dane**:

```
# Jeśli nie chcesz pobierać pliku na dysk możesz go wczytać bezpośrednio do R
# za pomocą poniższego kodu:
dane <- readRDS(gzcon(url("http://enwo.pl/przetwarzanie/dane/synop.rds")))

# ... Przyjrzyjmy się strukturze wczytanej bazy:
```

```
head(dane)
```

```
##      kod nazwa  yy mm dd hh  t2m ws  wd    slp tot_cl
## 1 352160330  POZ 2000  1  1  0 -1.1  3 250 1014.2      8
## 2 352160330  POZ 2000  1  1  1 -1.1  2 280 1014.2      8
## 3 352160330  POZ 2000  1  1  2 -1.1  2 250 1014.3      8
## 4 352160330  POZ 2000  1  1  3 -1.2  2 240 1014.3      8
## 5 352160330  POZ 2000  1  1  4 -1.2  2 200 1014.3      8
## 6 352160330  POZ 2000  1  1  5 -1.0  1 170 1014.3      8
```

```
# oraz:
```

```
summary(dane)
```

```
##      kod      nazwa      yy      mm
## Length:420766 Length:420766 Min.   :2000 Min.   : 1.000
## Class :character Class :character 1st Qu.:2004 1st Qu.: 4.000
## Mode  :character Mode  :character Median :2008 Median : 7.000
##                                     Mean  :2007 Mean  : 6.523
##                                     3rd Qu.:2012 3rd Qu.:10.000
##                                     Max.   :2015 Max.   :12.000
##      dd      hh      t2m      ws
## Min.   : 1.00 Min.   : 0.00 Min.   : -30.10 Min.   : 0.000
## 1st Qu.: 8.00 1st Qu.: 6.00 1st Qu.:  2.30 1st Qu.: 2.000
## Median :16.00 Median :12.00 Median :  9.20 Median : 3.000
## Mean   :15.73 Mean   :11.50 Mean   :  9.16 Mean   : 3.492
## 3rd Qu.:23.00 3rd Qu.:17.75 3rd Qu.: 15.90 3rd Qu.: 5.000
## Max.   :31.00 Max.   :23.00 Max.   : 36.90 Max.   :20.000
##      wd      slp      tot_cl
## Min.   : 0.0 Min.   : 0.08 Min.   :0.000
## 1st Qu.:101.0 1st Qu.: 993.80 1st Qu.:3.000
## Median :200.0 Median :1000.70 Median :6.000
## Mean   :190.3 Mean   :1000.39 Mean   :5.191
## 3rd Qu.:270.0 3rd Qu.:1007.20 3rd Qu.:7.000
## Max.   :999.0 Max.   :1038.90 Max.   :9.000
```


9.1 Wybór kolumn - `select()`

Niejednokrotnie wczytywane przez nas zbiory danych meteorologicznych zawierają znacznie więcej kolumn niż potrzebujemy. Jeśli chcemy wybrać lub pozbyć się niektórych kolumn najwygodniej użyć funkcji `select()`.

Korzystanie z funkcji `select()` wymaga podania jako pierwszego argumentu nazwy zbioru, a następnie po przecinku (bez `*` " ") należy podać nazwy kolumn (w dowolnej kolejności) które mają zostać wyświetlone.

Przykład: Jeśli chcemy wybrać jedynie wartości z nazwą stacji, rokiem, miesiącem, dniem, godziną, temperaturą powietrza oraz zachmurzeniem składnia takiego polecenia wyglądała by następująco:

```
library(dplyr) # nie zapomnijmy o aktywacji paczki dplyr po uruchomieniu RStudio!  
test <- select(dane, nazwa, yy, mm, dd, hh, t2m, tot_cl)  
head(test)
```

```
##   nazwa   yy mm dd hh  t2m tot_cl  
## 1   POZ 2000  1  1  0 -1.1      8  
## 2   POZ 2000  1  1  1 -1.1      8  
## 3   POZ 2000  1  1  2 -1.1      8  
## 4   POZ 2000  1  1  3 -1.2      8  
## 5   POZ 2000  1  1  4 -1.2      8  
## 6   POZ 2000  1  1  5 -1.0      8
```

W powyższym przykładzie konieczne było podanie aż 7 z 11 nazw kolumn. W takim przypadku można wykorzystać opcję eliminacji wybranych kolumn poprzez zastosowanie znaku minusa (-) w odniesieniu do kolumn których chcemy się pozbyć. Taki sam efekt jak we wcześniejszym przykładzie można uzyskać zatem poprzez:

Przykład:

```
test <- select(dane, -kod, -ws, -wd, -slp)  
head(test)
```

```
##   nazwa   yy mm dd hh  t2m tot_cl  
## 1   POZ 2000  1  1  0 -1.1      8  
## 2   POZ 2000  1  1  1 -1.1      8
```

```
## 3 POZ 2000 1 1 2 -1.1      8
## 4 POZ 2000 1 1 3 -1.2      8
## 5 POZ 2000 1 1 4 -1.2      8
## 6 POZ 2000 1 1 5 -1.0      8
```

Kolejne ciekawe zastosowane `selecta` polega na zastosowaniu jako operatora : (dwukropka), który wybierze wszystkie kolumny znajdujące się w zakresie od wskazanej do wskazanej nazwy. Wcześniejsze poleceni

Przykład:

```
test <- select(dane, nazwa:hh,t2m, tot_cl)
head(test)
```

```
##   nazwa   yy mm dd hh  t2m tot_cl
## 1 POZ 2000  1  1  0 -1.1      8
## 2 POZ 2000  1  1  1 -1.1      8
## 3 POZ 2000  1  1  2 -1.1      8
## 4 POZ 2000  1  1  3 -1.2      8
## 5 POZ 2000  1  1  4 -1.2      8
## 6 POZ 2000  1  1  5 -1.0      8
```

Więcej praktycznych przykładów zastosowania pakietu `dplyr` można znaleźć w rzeczonym we wstępie do niniejszego rozdziału *cheat-sheetcie* w podrozdziale *Helper functions for select...* .

Zadanie:

1. Wybierz ze zbioru `dane` tylko kolumny dla nazwy, kodu, roku, miesiąca oraz zredukowanego ciśnienia atmosferycznego i zapisz je do zbioru `test`
2. Wybierz ze zbioru `dane` tylko kolumny z nazwami stacji oraz wszystkich parametrów meteorologicznych. Zastosuj zapis negacji (tj. z wykorzystaniem znaku minusa) dla usuwanych kolumn i zapisz wynik działania jako `test2`

9.2 Filtrowanie

Duże zbiory danych meteorologicznych wymagają odfiltrowania (usunięcia/wybrania) części informacji zapisanych w wierszach. Odfiltrowywanie danych jest w wielu przypadkach wymagane np. w celu znalezienia i eliminacji błędów z bazy danych lub uzyskania kształtu ramki danych tylko dla interesujących nas przypadków.

Do wyświetlania/usuwania wybranych wierszy (oprócz poznanych wcześniej operatorów `[]` oraz operatorów zapytań logicznych) wygodne w użyciu może okazać się wykorzystanie funkcji `filter()`.

Przykład 1 Baza `dane` zawiera wartości pomiarowe dla kilku stacji meteorologicznych. Informacje o tym jakie to są stacje można sprawdzić wyświetlając np. unikalne wartości kolumny `kod` lub `nazwa`.

```
unique(dane$nazwa) # wyświetla wartości unikalne z podanego wektora
```

```
## [1] "POZ" "WAR" "LOD"
```

Okazuje się, że w bazie są dane dla 3 stacji IMGW-PIB, które oznaczono w bazie 3 literowymi skrótami: Poznań (POZ), Warszawa (WAR) oraz ŁÓDŹ (LOD). Jeśli do naszej dalszej pracy potrzebna jest tylko 1 stacja (np. Poznań), wówczas możemy odfiltrować wiersze tylko do tych zawierających słowo “POZ” w kolumnie `nazwa`:

```
test <- filter(dane, nazwa=="POZ")
head(test)
```

```
##      kod nazwa  yy mm dd hh  t2m ws  wd    slp tot_cl
## 1 352160330   POZ 2000  1  1  0 -1.1  3 250 1014.2     8
## 2 352160330   POZ 2000  1  1  1 -1.1  2 280 1014.2     8
## 3 352160330   POZ 2000  1  1  2 -1.1  2 250 1014.3     8
## 4 352160330   POZ 2000  1  1  3 -1.2  2 240 1014.3     8
## 5 352160330   POZ 2000  1  1  4 -1.2  2 200 1014.3     8
## 6 352160330   POZ 2000  1  1  5 -1.0  1 170 1014.3     8
```

Zwróć uwagę, że nazwę kolumny (`nazwa`) ponownie wpisaliśmy bez cudzysłowów, natomiast poszukiwana wartość (POZ) jest tekstem, więc tym razem konieczne było zastosowanie `" "`.

Przykład 2 Stosowane operatory logiczne dla funkcji `filter()` są tożsame z poznanymi we wcześniejszych częściach naszego kursu. Jeśli chcemy jednocześnie odfiltrować np. tylko wiersze, które zawierają w kolumnie nazwa wartości "POZ" i jednocześnie obejmują miesiące meteorologicznego lata (VI-VIII), to składnia takiego polecenia może być następująca:

```
test <- filter(dane, nazwa=="POZ", mm>=6 & mm<=8)
head(test)
```

##		kod	nazwa	yy	mm	dd	hh	t2m	ws	wd	slp	tot_cl
## 1	352160330	POZ	2000	6	1	0	4.6	1	280	1013.2	0	
## 2	352160330	POZ	2000	6	1	1	4.6	0	0	1013.6	2	
## 3	352160330	POZ	2000	6	1	2	3.3	0	0	1013.7	2	
## 4	352160330	POZ	2000	6	1	3	3.5	1	360	1014.0	3	
## 5	352160330	POZ	2000	6	1	4	6.2	1	90	1014.3	3	
## 6	352160330	POZ	2000	6	1	5	9.7	2	100	1014.6	3	

Przykład 3 Powyższy przykład z wyborem więcej niż jednej pasującej wartości w danej kolumnie nie zawsze musi być tak trywialny aby dało się go rozpisać za pomocą pojedynczego wyrażenia logicznego. W takich przypadkach można zastosować odfiltrowywanie wierszy za pomocą porównania więcej niż jednego elementu, który musi zostać zadeklarowany jako wektor po operatorze `%in%`.

Jeśli interesują nas tylko wiersze obejmujące miesiące zimowe (XII-II) i tylko z Poznania (POZ) oraz Łodzi możemy spróbować działanie poniższej komendy:

```
test <- filter(dane, nazwa %in% c("POZ", "LOD"), mm %in% c(12,1,2))
head(test)
```

##		kod	nazwa	yy	mm	dd	hh	t2m	ws	wd	slp	tot_cl
## 1	352160330	POZ	2000	1	1	0	-1.1	3	250	1014.2	8	
## 2	352160330	POZ	2000	1	1	1	-1.1	2	280	1014.2	8	
## 3	352160330	POZ	2000	1	1	2	-1.1	2	250	1014.3	8	
## 4	352160330	POZ	2000	1	1	3	-1.2	2	240	1014.3	8	
## 5	352160330	POZ	2000	1	1	4	-1.2	2	200	1014.3	8	
## 6	352160330	POZ	2000	1	1	5	-1.0	1	170	1014.3	8	

```
unique(test$mm) # sprawdzmy czy na pewno sa tylko miesiace zimowe
```

```
## [1] 1 2 12
```

```
unique(test$nazwa) # sprawdzmy czy na pewno sa tylko wybrane stacje
```

```
## [1] "POZ" "LOD"
```

Przykład 4 Rozpoznanie błędów w bazach danych meteorologicznych nie należy do zadań łatwych. Część błędów można jednak dość łatwo rozpoznać znając fizyczne ograniczenia występujących wartości. Sprawdźmy naszą bazę pod tym kątem na przykładzie zachmurzenia ogólnego nieba, które jest podawane w oktantach (0-8). Jeśli chcemy sprawdzić czy istnieją wartości nie mieszczące się w tym zakresie możemy spróbować zdefiniować takie zapytanie logiczne, które umożliwi nam zweryfikowanie takich informacji i zapisanie ich do nowego obiektu:

```
bledy <- filter(dane, tot_cl<0 | tot_cl>8)
head(bledy)
```

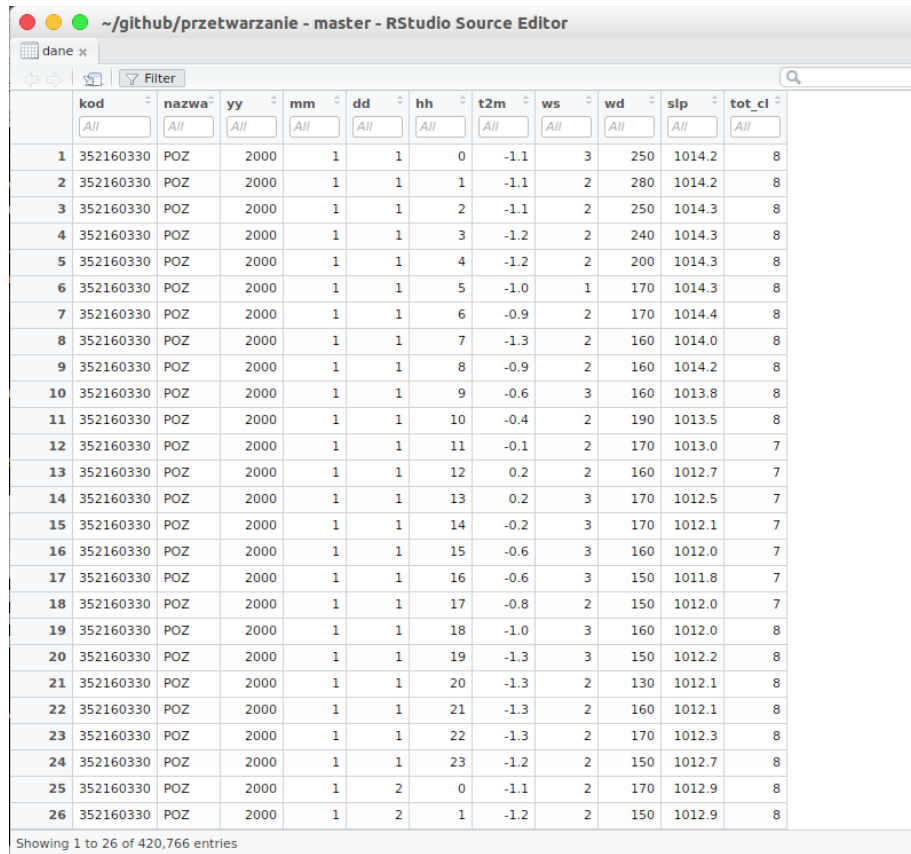
```
##          kod nazwa   yy mm dd hh  t2m ws  wd   slp tot_cl
## 1 352160330    POZ 2000  3 23  5 -1.6  1 210 1004.5     9
## 2 352160330    POZ 2000  3 26  1  3.4  1 190 1003.1     9
## 3 352160330    POZ 2000  3 26  2  3.4  1 200 1002.9     9
## 4 352160330    POZ 2000  9  5 23  7.8  0   0 1008.6     9
## 5 352160330    POZ 2000  9  6  0  8.7  1 350 1008.6     9
## 6 352160330    POZ 2000  9  6  1  7.3  0   0 1008.6     9
```

**** Zadanie ****

1. Znajdź wiersze w których ciśnienie atmosferyczne zawiera błędne dane
2. Odfiltruj bazę danych w taki sposób aby zawierały tylko dane dla Poznania i Warszawy. Wynik zapisz do nowego obiektu `test1`
3. Odfiltruj bazę danych `test1` w taki sposób aby nowa baza `test2` zawierała tylko dane dla Poznania, z miesięcy letnich po roku 2005 4a. Oblicz (a) średnią, (b) minimalną i (c) maksymalną temperaturę powietrza jaka wystąpiła w Poznaniu w tym czasie. 4b. Oblicz (a) średnią (b) oraz maksymalną prędkość wiatru

*Podpowiedź: Szybkie filtrowanie danych jest dostępne również za pomocą graficznej przeglądarki RStudio po kliknięciu w ikonę **Filter** (ale bez możliwości zapisania wykonanego zapytania do pliku). W trybie graficznym możliwe jest także sortowanie wierszy po kliknięciu (jedno- lub dwukrotnym) w nazwę danej*

kolumny.



	kod	nazwa	yy	mm	dd	hh	t2m	ws	wd	slp	tot_cl
1	352160330	POZ	2000	1	1	0	-1.1	3	250	1014.2	8
2	352160330	POZ	2000	1	1	1	-1.1	2	280	1014.2	8
3	352160330	POZ	2000	1	1	2	-1.1	2	250	1014.3	8
4	352160330	POZ	2000	1	1	3	-1.2	2	240	1014.3	8
5	352160330	POZ	2000	1	1	4	-1.2	2	200	1014.3	8
6	352160330	POZ	2000	1	1	5	-1.0	1	170	1014.3	8
7	352160330	POZ	2000	1	1	6	-0.9	2	170	1014.4	8
8	352160330	POZ	2000	1	1	7	-1.3	2	160	1014.0	8
9	352160330	POZ	2000	1	1	8	-0.9	2	160	1014.2	8
10	352160330	POZ	2000	1	1	9	-0.6	3	160	1013.8	8
11	352160330	POZ	2000	1	1	10	-0.4	2	190	1013.5	8
12	352160330	POZ	2000	1	1	11	-0.1	2	170	1013.0	7
13	352160330	POZ	2000	1	1	12	0.2	2	160	1012.7	7
14	352160330	POZ	2000	1	1	13	0.2	3	170	1012.5	7
15	352160330	POZ	2000	1	1	14	-0.2	3	170	1012.1	7
16	352160330	POZ	2000	1	1	15	-0.6	3	160	1012.0	7
17	352160330	POZ	2000	1	1	16	-0.6	3	150	1011.8	7
18	352160330	POZ	2000	1	1	17	-0.8	2	150	1012.0	7
19	352160330	POZ	2000	1	1	18	-1.0	3	160	1012.0	8
20	352160330	POZ	2000	1	1	19	-1.3	3	150	1012.2	8
21	352160330	POZ	2000	1	1	20	-1.3	2	130	1012.1	8
22	352160330	POZ	2000	1	1	21	-1.3	2	160	1012.1	8
23	352160330	POZ	2000	1	1	22	-1.3	2	170	1012.3	8
24	352160330	POZ	2000	1	1	23	-1.2	2	150	1012.7	8
25	352160330	POZ	2000	1	2	0	-1.1	2	170	1012.9	8
26	352160330	POZ	2000	1	2	1	-1.2	2	150	1012.9	8

Showing 1 to 26 of 420,766 entries

Rysunek 9.1: Ekran początkowy programu **RStudio**

9.3 Sortowanie - arrange

Sortowanie wierszy w **R** możliwe jest na co najmniej kilka sposobów. W pakiecie **dplyr** funkcja odpowiedzialna za sortowanie ramek danych to **arrange()**. Działa ona bardzo podobnie do **select()**, gdzie na pierwszym miejscu jako argument podaje się nazwę ramki danych, a w kolejnych nazwy kolumn.

Przykład 1 Wyświetlmy kilka najchłodniejszych pomiarów (kolumna **t2m**) z całej bazy (domyślnie sortowanie odbywa się od wartości najmniejszych do największych):

```
test <- arrange(dane, t2m)
head(test)
```

```
##          kod nazwa   yy mm dd hh   t2m ws  wd    slp tot_cl
## 1 351190465   LOD 2006  1 23  6 -30.1  0   0 1023.0      0
## 2 351190465   LOD 2006  1 23  3 -29.7  1 180 1022.0      0
## 3 351190465   LOD 2006  1 23  5 -29.7  0   0 1023.0      0
## 4 351190465   LOD 2006  1 23  2 -29.5  0   0 1022.0      0
## 5 351190465   LOD 2006  1 23  7 -29.5  0   0 1023.5      0
## 6 351190465   LOD 2006  1 22 23 -28.8  1 240 1021.0      0
```

Przykład 2 Jeśli chcemy zmienić domyślną (narastającą) kolejność sortowania możemy odwrócić kierunek sortowania za pomocą znaku minusa przy nazwie kolumny. Sprawdźmy zatem kilka rekordowych wartości temperatur powietrza zanotowanych na analizowanych stacjach:

```
test <- arrange(dane, -t2m)
head(test)
```

```
##          kod nazwa   yy mm dd hh   t2m ws  wd    slp tot_cl
## 1 351190465   LOD 2013  8  8 14 36.9  6 204  989.8      2
## 2 352160330   POZ 2015  8  8 14 36.8  1 137 1004.9      1
## 3 352200375   WAR 2013  8  8 13 36.7  4 152 1000.0      2
## 4 352200375   WAR 2013  8  8 15 36.7  2 191  999.8      4
## 5 352200375   WAR 2013  8  8 14 36.6  2 185 1000.0      4
## 6 351190465   LOD 2013  8  8 13 36.5  7 208  989.8      4
```

```
# okazuje sie, ze wszedzie najcieplejszy byl 8. sierpnia, choc...
```

**** Przykład 3 ****

W przypadku w którym wartości sortowane mają takie same wartości często wykorzystywaną procedurą jest sortowanie po większej liczbie kolumn. Spróbujmy zatem ułożyć dane nie w kolejności dla stacji, ale po datach. Jako że wartości dla dat przechowywane są w 4 kolumnach (od “yy” do “hh”) ważne jest określenie odpowiedniej kolejności sortowania:

```
test <- arrange(dane, yy,mm,dd,hh,nazwa)
head(test)
```

```
##          kod nazwa  yy mm dd hh  t2m ws  wd    slp tot_cl
## 1 352160330  POZ 2000  1  1  0 -1.1  3 250 1014.2      8
## 2 351190465  LOD 2000  1  1  1 -1.8  2 270 1001.6      8
## 3 352160330  POZ 2000  1  1  1 -1.1  2 280 1014.2      8
## 4 352200375  WAR 2000  1  1  1  0.0  2 280 1011.2      8
## 5 351190465  LOD 2000  1  1  2 -1.7  1 250 1001.5      8
## 6 352160330  POZ 2000  1  1  2 -1.1  2 250 1014.3      8
```

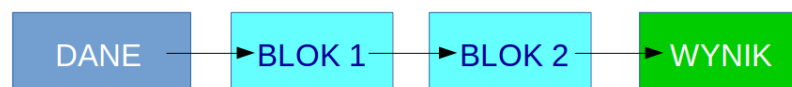
**** Zadanie ****

1. Posortuj dane w kolejności od największych do najmniejszych prędkości wiatru. Zapisz wynik sortowania do obiektu `test`
2. Korzystając z komendy `table()` policz ile razy każda ze stacji pojawia się w zestawieniu 30-tu pomiarów z największymi prędkościami wiatru

9.4 Przetwarzanie potokowe

Pakiet `dplyr` zrewolucjonizował przetwarzanie danych w środowisku **R** na wiele sposobów. Jednym z nich jest zastosowanie przetwarzania potokowego (zwanego również przetwarzaniem sekwencyjnym), które nie tylko zwiększa czytelność tworzonego kodu, ale także pozwala na pominięcie kroków pośrednich, które normalnie *zapychałyby* środowisko obliczeniowe.

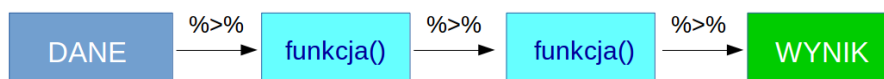
W przetwarzaniu potokowym cykl przetwarzania dzieli się na odrębne bloki, z których każdy jest połączony z następnym. Dane po przejściu przez jeden blok trafiają do następnego, aż osiągną ostatni blok.



Rysunek 9.2: Schemat typowego przebiegu przetwarzania potokowego przy analizie danych

Operatorem przetwarzania potokowego w **R** jest `%>%`, który można wygenero-

wać w *RStudio* za pomocą skrótu klawiszowego `ctrl+shift+m`. Operator `%>%` przekierowuje strumień informacji do kolejnej komendy **R**, która działając na tymczasowym obiekcie przekazuje jej wynik do kolejnego bloku obliczeń. Jeśli zaistnieje konieczność odwołania się do informacji zawartych w strumieniu danych można się do niego odwołać poprzez symbol `.` (kropki).



Rysunek 9.3: Przetwarzanie potokowego w **R**

Przykład 1 Na początek przypomnijmy sobie zawartość wbudowanego zbioru `airquality`

```
head(airquality)
```

```
##      Ozone Solar.R Wind Temp Month Day      TempC
## 1      41      190  7.4  100     5   1 37.77778
## 2      36      118  8.0   72     5   2 22.22222
## 3      12      149 12.6   74     5   3 23.33333
## 4      18      313 11.5   62     5   4 16.66667
## 5      NA       NA 14.3   56     5   5 13.33333
## 6      28       NA 14.9   66     5   6 18.88889
```

Ten sam efekt możemy uzyskać stosując przetwarzanie potokowe, wiedząc, że każda `funkcja(x)` w postaci przetwarzania potokowego to `x %>% funkcja()`, zatem:

```
airquality %>% head() #lub: airquality %>% head(.)
```

```
##      Ozone Solar.R Wind Temp Month Day      TempC
## 1      41      190  7.4  100     5   1 37.77778
## 2      36      118  8.0   72     5   2 22.22222
## 3      12      149 12.6   74     5   3 23.33333
## 4      18      313 11.5   62     5   4 16.66667
## 5      NA       NA 14.3   56     5   5 13.33333
## 6      28       NA 14.9   66     5   6 18.88889
```

Przykład 2 Jeśli chcemy wybrać jedynie dni, w których (1) temperatura powietrza przekroczyła 90F, (2) chcemy się pozbyć niechcianych kolumn pozostawiając jedynie kolumny Temp i Month i Day, (3) chcemy posortować te dni w kolejności od najcieplejszych do najchłodniejszych, to ten efekt przy klasycznym przetwarzaniu danych byłby rozpisany w kilku krokach:

```
krok1 <- filter(airquality, Temp>90) # wybieramy wiersza w których temperatura powietrza
krok2 <- select(krok1, Temp:Day) # wybieramy tylko wskazane kolumny
krok3 <- arrange(krok2, -Temp) # sortujemy od najwyższych do najniższych temperatur
```

Teoretycznie ten sam efekt można uzyskać w jednym kroku, stosując np. tzw. zapis na *cebulkę*, choć jego czytelność pozostawia wiele do życzenia:

```
arrange(
  select(
    filter(airquality, Temp>90), # pierwsza funkcja
    Temp:Day), # dokończenie funkcji select
  -Temp) # dokończenie funkcji arrange
```

```
##      Temp Month Day
## 1    100     5   1
## 2     97     8  28
## 3     96     8  30
## 4     94     8  29
## 5     94     8  31
## 6     93     6  11
## 7     93     9   3
## 8     93     9   4
## 9     92     6  12
## 10    92     7   8
## 11    92     7   9
## 12    92     8  10
## 13    92     9   2
## 14    91     7  14
## 15    91     9   1
```

Przy zastosowaniu przetwarzania potokowego:

```
airquality %>% filter(Temp>90) %>% select(Temp:Day) %>% arrange(-Temp)
```

```
##      Temp Month Day
## 1    100     5   1
## 2     97     8  28
## 3     96     8  30
## 4     94     8  29
## 5     94     8  31
## 6     93     6  11
## 7     93     9   3
## 8     93     9   4
## 9     92     6  12
## 10    92     7   8
## 11    92     7   9
## 12    92     8  10
## 13    92     9   2
## 14    91     7  14
## 15    91     9   1
```

```
# lub w wersji z kropką: airquality %>% filter(., Temp>90) %>% select(., Temp:Day) %>% arrange(.,
```

Zadanie:

1. Pobierz ponownie dane z pliku <http://enwo.pl/przetwarzanie/dane/synop.rds> i zapisz jako obiekt `dane` (możesz także wykorzystać gotowy kod: `dane <- readRDS(gzcon(url("http://enwo.pl/przetwarzanie/dane/synop.rds")))`)
2. Korzystając z przetwarzania potokowego wybierz jedynie dane meteorologiczne dla Poznania
3. W kolejnym bloku kodu wybierz jedynie miesiące letnie (VI-VIII)
4. W kolejnym bloku kodu uszereguj wyniki od najniższych temperatur jakie odnotowano latem
5. Zapisz rezultat działania do nowego obiektu, który nazwiesz `wynik`

9.5 group_by() oraz summarise()

Z punktu widzenia przetwarzania danych w naukach atmosferycznych niezwykle ważna jest możliwość szybkiego tworzenia tzw. agregatów, czyli tworzenia podsumowań dla analizowanych zbiorów danych. Do tego celu niezwykle przydatne są 2 funkcje z pakietu dplyr: `group_by()` oraz `summarise()`.

Jak sama nazwa wskazuje funkcja `group_by()` grupuje zbiory danych po unikalnych wartościach we wskazanych kolumnach ramki danych. Przykładowo, jeśli wskażemy do grupowania kolumny `yy`, `mm`, `dd`, wówczas będą one grupowały poszczególne wiersze w ramce danych jak na wskazanym poniżej przykładzie:

	kod	nazwa	yy	mm	dd	hh	t2m	ws	wd	slp	tot_cl
1	352160330	POZ	2000	1	1	0	-1.1	3	250	1014.2	8
2	352160330	POZ	2000	1	1	3	-1.2	2	240	1014.3	8
3	352160330	POZ	2000	1	1	6	-0.9	2	170	1014.4	8
4	352160330	POZ	2000	1	1	9	-0.6	3	160	1013.8	8
5	352160330	POZ	2000	1	1	12	0.2	2	160	1012.7	7
6	352160330	POZ	2000	1	1	15	-0.6	3	160	1012.0	7
7	352160330	POZ	2000	1	1	18	-1.0	3	160	1012.0	8
8	352160330	POZ	2000	1	1	21	-1.3	2	160	1012.1	8
9	352160330	POZ	2000	1	2	0	-1.1	2	170	1012.9	8
10	352160330	POZ	2000	1	2	3	-1.6	2	160	1013.4	8
11	352160330	POZ	2000	1	2	6	-1.4	2	160	1014.5	8
12	352160330	POZ	2000	1	2	9	-0.9	2	210	1015.5	8
13	352160330	POZ	2000	1	2	12	0.8	2	240	1015.2	8
14	352160330	POZ	2000	1	2	15	1.3	2	230	1015.7	8
15	352160330	POZ	2000	1	2	18	1.5	3	250	1014.6	8
16	352160330	POZ	2000	1	2	21	1.5	5	240	1014.3	8
17	352160330	POZ	2000	1	3	0	2.0	6	250	1013.1	8
18	352160330	POZ	2000	1	3	3	1.8	5	260	1012.0	8
19	352160330	POZ	2000	1	3	6	2.2	4	260	1012.2	8
20	352160330	POZ	2000	1	3	9	2.6	6	250	1011.8	8
21	352160330	POZ	2000	1	3	12	3.0	5	250	1011.1	8
22	352160330	POZ	2000	1	3	15	2.9	5	230	1009.3	8
23	352160330	POZ	2000	1	3	18	3.2	4	230	1008.5	8
24	352160330	POZ	2000	1	3	21	3.3	4	230	1008.1	8

Rysunek 9.4: Zasięg oddziaływania funkcji `group_by()` przy wskazaniu jako grupujących kolumn `yy`, `mm`, `dd`

Samo grupowanie nie daje widocznych efektów. Jego wynik może być jednak z powodzeniem wykorzystany przez funkcję `summarise()`, która pozwala na wykonanie dowolnej funkcji w obrębie wydzielonych zagregowań. Takie postępowanie można z powodzeniem wykorzystać do obliczenia podstawowych statystyk meteorologicznych/klimatologicznych.

Przykład 1 Poniższy fragment kodu grupujący po kolumnie `yy` (lata) pozwoli na zgrupowanie wszystkich wierszy, które zawierają takie same wartości w tej kolumnie (tu: rok). Następnie dane są przekazywane do funkcji `summarise()`, w której obliczamy średnie z kolumny z wartościami temperatury:

```
dane %>% group_by(yy) %>% summarise( mean(t2m) )
```

```
## # A tibble: 16 x 2
##       yy `mean(t2m)`
##   <int>     <dbl>
## 1  2000      9.76
## 2  2001      8.47
## 3  2002      9.38
## 4  2003      8.61
## 5  2004      8.66
## 6  2005      8.80
## 7  2006      9.20
## 8  2007      9.71
## 9  2008      9.85
##10  2009      8.92
##11  2010      7.77
##12  2011      9.30
##13  2012      8.96
##14  2013      8.87
##15  2014     10.1
##16  2015     10.2
```

Czy uzyskany zbiór danych zawiera informacje dla temperatury średniej rocznej w Poznaniu? **Nie!** W ramce danych mamy pomiary z kilku stacji meteorologicznych. Z tego względu bez względu czy pomiar był w Poznaniu czy Łodzi, czy w Warszawie wszystkie pomiary z danego roku zostały wrzucone do tej samej

grupy (agregaty). W takim przypadku konieczne jest albo wcześniejsze odfiltrowania zbioru danych, albo rozszerzenie argumentów funkcji `group_by()`, tak aby wykonała obliczenia dla większej liczby stacji, np.:

```
dane %>% group_by(nazwa, yy) %>% summarise( mean(t2m) )
```

```
## # A tibble: 48 x 3
## # Groups:   nazwa [?]
##   nazwa    yy `mean(t2m)`
##   <chr> <int>      <dbl>
## 1 LOD    2000        9.60
## 2 LOD    2001        8.13
## 3 LOD    2002        9.10
## 4 LOD    2003        8.53
## 5 LOD    2004        8.53
## 6 LOD    2005        8.64
## 7 LOD    2006        8.87
## 8 LOD    2007        9.35
## 9 LOD    2008        9.58
## 10 LOD   2009        8.63
## # ... with 38 more rows
```

Zadanie:

1. Oblicz średnie miesięczne temperatury powietrza w wieloleciu 2000-2015 w Poznaniu (czyli np. I:-0.2°C, II:-2.1, III:+2.6, itd.)
2. Oblicz maksymalne temperatury jakie wystąpiły w każdym z miesięcy w wieloleciu 2000-2015 w Poznaniu (czyli np. 2000-I: +2.8°C, 2000-II:+6.1, 2000-III:+12.6, itd.)
3. Oblicz sumę temperatur w każdym dniu kalendarzowym w Poznaniu (analogiczną procedurę można by było zastosować np. do opadów atmosferycznych)

Wskazówka: Nazwy kolumn w funkcji `summarise()` można dowolnie definiować. Przykładowo, jeśli chcielibyśmy obliczyć jednocześnie temperaturę maksymalną i minimalną dobową w Poznaniu latem 2015 r., wówczas kolumny wynikowe możemy odpowiednio nazwać: ‘

```
dane %>% filter(., nazwa=="P0Z", yy==2015, mm %in% c(6:8)) %>% group_by(yy,mm,dd) %>% summarise
```

```
## # A tibble: 92 x 5
## # Groups:   yy, mm [?]
##       yy     mm     dd temp_min temp_max
##   <int> <int> <int>   <dbl>   <dbl>
## 1  2015     6     1    12.5    22.6
## 2  2015     6     2    10.7    24.9
## 3  2015     6     3    12.1    28.4
## 4  2015     6     4     9.2    21.7
## 5  2015     6     5     9.7    26.1
## 6  2015     6     6    15.2    31.6
## 7  2015     6     7    11.8    18.2
## 8  2015     6     8     9.2    20.5
## 9  2015     6     9     8.2    14.8
##10  2015     6    10     8.6    20.7
## # ... with 82 more rows
```

Zadanie sprawdzające

Rozdział 10

Postać wąska i szeroka

Przechowywanie danych wiąże się nie tylko z różnymi formatami zapisu danych, ale także z różnymi możliwościami rozłożenia tych samych informacji na wiersze i kolumny. W analizie danych rozróżnia się dwa podstawowe typy przechowywania informacji: postać wąską i szeroką. *“Po co taka różnorodność? Otóż w zależności od tego co z danymi chcemy zrobić czasem lepiej je mieć w takiej czy innej postaci”* (Biecek, 2016)

Przejście z jednej postaci do drugiej jest dość często wykonywaną operacją (nie tylko w **R**), choć przez wielu uznawaną za operację czasochłonną. Rozwiązaniem są funkcje pakietu `tidyr`.

10.1 Postać wąska

Dane w postaci wąskiej mogą przypominać swoim kształtem rozwiązania bazodanowe (np. **SQL** i pokrewne). Przykładowa baza danych zawiera 6-godz. sumy opadów atmosferycznych pobranych ze strony <https://dane.imgw.pl>. Dane dla kilku stacji zapisano do postaci wąskiej w pliku `.rds` udostępnionym pod adresem: <http://enwo.pl/przetwarzanie/dane/opady.rds>

1. Wczytaj plik do środowiska **R** i nazwij go `dane`.
2. Zapoznaj się z jego strukturą za pomocą funkcji: `summary()` i `str()`
3. Sprawdź dla ilu stacji dostępne są historyczne dane opadowe?

```
dane <- readRDS(gzcon(url("http://enwo.pl/przetwarzanie/dane/opady.rds")))

# ... Przyjrzyjmy się strukturze wczytanej bazy:
head(dane)

##              data wartosc stacja
## 1 1966-01-01 00:00:00      NA 12560
## 2 1966-01-01 06:00:00    0.4 12560
## 3 1966-01-01 12:00:00      NA 12560
## 4 1966-01-01 18:00:00    0.0 12560
## 5 1966-01-02 00:00:00    1.1 12560
## 6 1966-01-02 06:00:00      NA 12560

# lub:
str(dane)

## 'data.frame':    220111 obs. of  3 variables:
## $ data      : POSIXct, format: "1966-01-01 00:00:00" "1966-01-01 06:00:00" ...
## $ wartosc: num  NA 0.4 NA 0 1.1 NA NA NA 0.8 NA ...
## $ stacja : num 12560 12560 12560 12560 12560 ...

# Policzmy ile wartości jest dla każdej stacji
table(dane$stacja)

##
## 12415 12520 12560
## 74543 73655 71913
```

Konwersja postaci wąskiej do szerokiej - `spread()`

Jest to typowy przykład wąskiej postaci bazy danych, który bardzo dobrze sprawdza się np. w rozwiązaniach z grupowaniem i agregowaniem wartości (`group_by()` i `summarise()`). Jeśli chcielibyśmy “przenieść” wartości dla stacji, tak aby znajdowały się one w kolumnach obok siebie możemy skorzystać z funkcji `spread()`.

Funkcja `spread()` wymaga zadeklarowania 3 argumentów: **data** - zbioru danych, **key** - nazwy kolumny, która zostanie przetransformowana jako nazwy nowych

nagłówków kolumn, `value` - nazwa kolumny z wartościami, którymi zostanie wypełniona tabela.

Sprawdźmy działanie tej funkcji na przykładzie zbioru `dane`:

```
library(tidyr) # musimy pakiet aktywować / instalować jeśli uruchamiany po raz pierwszy
```

```
head(dane) # przypomnienie kształtu bazy
```

```
##           data wartosc stacja
## 1 1966-01-01 00:00:00      NA 12560
## 2 1966-01-01 06:00:00    0.4 12560
## 3 1966-01-01 12:00:00      NA 12560
## 4 1966-01-01 18:00:00    0.0 12560
## 5 1966-01-02 00:00:00    1.1 12560
## 6 1966-01-02 06:00:00      NA 12560
```

```
wynik <- spread(dane, stacja, wartosc) # konwertujemy do postaci szerokiej
head(wynik) # i sprawdzamy otrzymany wynik
```

```
##           data 12415 12520 12560
## 1 1966-01-01 00:00:00  0.0    NA    NA
## 2 1966-01-01 06:00:00  NA    NA    0.4
## 3 1966-01-01 12:00:00  0.0    NA    NA
## 4 1966-01-01 18:00:00  0.9    0.0    0.0
## 5 1966-01-02 00:00:00  1.1    1.3    1.1
## 6 1966-01-02 06:00:00  NA    NA    NA
```

Zadania

1. Sprawdź co się stanie jeśli odwrócisz kolejność argumentów `key` i `value`?
2. Za pomocą funkcji `group_by()` oraz `summarise()` oblicz roczną sumę opadów na każdej ze stacji a następnie wynik tego działania przekonwertuj do dowolnej postaci szerokiej

10.2 Postać szeroka

Często bardziej intuicyjna w działaniu jest postać szeroka, która wizualnie pozwala na przeglądnięcie większej liczby danych. Czasem taki format danych może nie być zgodny np. z niektórymi funkcjami graficznymi, stąd konieczność transformacji z postaci szerokiej do wąskiej.

Konwersja postaci szerokiej do wąskiej - `gather()`

Wczytaj zbiór ze średnimi miesięcznymi temperaturami powietrza w Polsce po 1971 r.: <http://enwo.pl/przetwarzanie/dane/pl1.csv>. Nazwij zbiór `pl` i zapoznaj się z jego strukturą.

```
pl <- read.csv("http://enwo.pl/przetwarzanie/dane/pl1.csv")
head(pl)
```

```
##      rok      I      II     III     IV      V      VI      VII     VIII     IX      X      XI
## 1 1971 -2.96  0.37 -0.11  7.41 14.48 14.91 17.81 18.74 11.25 8.39 2.61
## 2 1972 -5.81  0.27  3.91  7.36 12.48 16.25 19.40 16.56 11.36 6.14 4.36
## 3 1973 -1.67  1.27  3.69  6.13 12.35 15.77 17.58 17.12 13.14 6.62 1.85
## 4 1974  0.06  2.29  4.37  6.89 10.75 14.04 15.62 17.70 13.40 6.29 3.86
## 5 1975  2.97 -0.56  3.87  6.58 13.35 15.40 18.55 18.27 15.79 7.98 1.75
## 6 1976 -1.97 -3.32 -0.85  6.72 11.95 15.09 18.06 15.31 12.58 7.59 4.71
##      XII
## 1  3.02
## 2 -0.01
## 3 -0.72
## 4  2.73
## 5  0.88
## 6 -1.24
```

W przypadku funkcji `gather()` służącej do konwertowania szerokiej postaci danych do wąskiej zestaw argumentów jest następujący: 1) `data` - ramka danych, 2) `key` oraz `value` - etykiety nowych kolumn z kluczem i wartościami, 3) `...` - ustalenie kolumn które mają zostać przekonwertowane według schematu dla paczek `dplyr/tidyr` (tj. bez `”`, z możliwością stosowania `:` dla kolumny początkowej i końcowej, itp.)

W naszym przypadku ramkę danych `pl` chcemy skonwertować tak, aby zawierała 3 kolumny: rok, miesiac, temperatura.

```
wynik <- gather(data=pl, key="miesiac", value="temperatura", I:XII)
head(wynik)
```

```
##      rok miesiac temperatura
## 1 1971      I      -2.96
## 2 1972      I      -5.81
## 3 1973      I      -1.67
## 4 1974      I       0.06
## 5 1975      I       2.97
## 6 1976      I      -1.97
```

10.3 Sklejanie i rozszczepianie kolumn

Ostatnie 2 ciekawe funkcje z pakietu `tidyr` to sklejanie (`unite()`) i rozszczepianie kolumn (`seperate()`). Ze względu na ograniczenia czasowe polecam zapoznanie się z tymi funkcjami indywidualnie

Pamiętaj o cheat-sheet'ie łączącym najważniejsze funkcje pakietów: dplyr i tidyr <https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

Rozdział 11

Grafika

Wstępne przetwarzanie danych jest tylko jednym z etapów pracy z danymi. Finalnym produktem przetwarzania są bardzo często wizualizacje tych danych w postaci graficznej. W przypadku **R** istnieje kilka silników graficznych, spośród których najbardziej popularny jest bazowy `graphics` oraz pakiet `ggplot2` i `lattice`.

Poniżej wymieniono

11.1 `graphics`

Podstawowy silnik graficzny **R**.

Funkcje:

- `plot()` + `lines()`

```
dane <- readRDS(gzcon(url("http://enwo.pl/przetwarzanie/dane/opady.rds")))
dane <- dane %>% group_by(rok=lubridate::year(dane$data), stacja) %>% summarise( suma=sum(wartos
dane <- filter(dane, rok<2017) # ostatni rok jest niepelny
dane2 <- filter(dane,stacja=="12415")
```

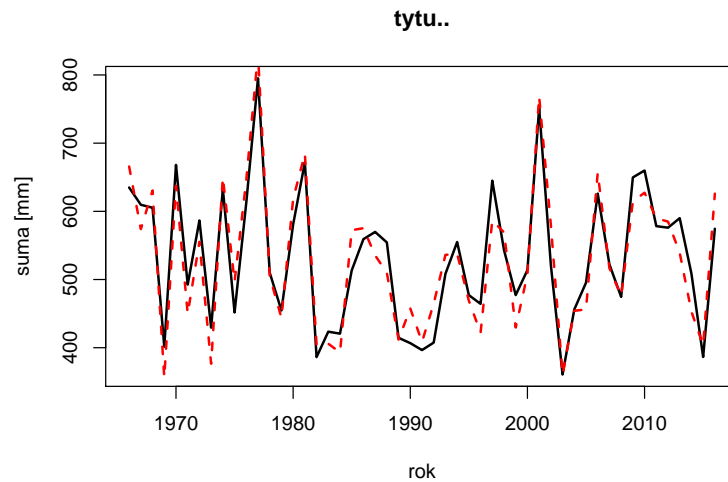
```
plot(x = dane2$rok, y = dane2$suma, type='l', main="tytuł", xlab='rok', ylab='suma [mm]', lwd=2)
```

```
## Warning in title(...): conversion failure on 'tytuł' in 'mbcsToSbcs': dot
```

```
## substituted for <c5>

## Warning in title(...): conversion failure on 'tytuł' in 'mbcsToSbcs': dot
## substituted for <82>

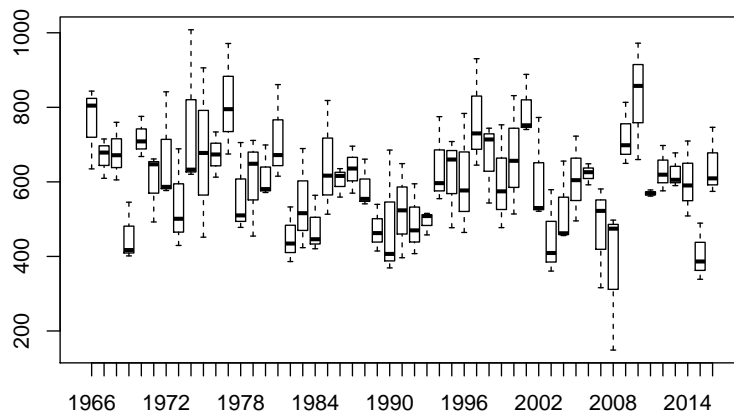
lines(x = dane2$rok, y = jitter(dane2$suma,3000), type='l', col="red", lty=2, lwd=2)
```



Rysunek 11.1: Przykład użycia funkcji plot i lines

- `boxplot()`

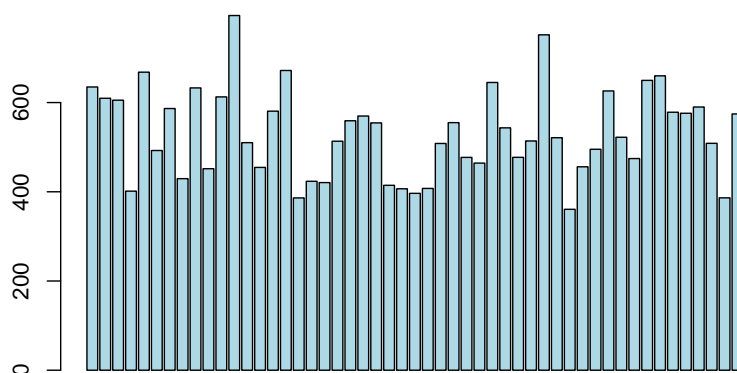
```
tidyr::spread(dane, rok, suma) %>% .[, -1] %>% boxplot()
```



Rysunek 11.2: Przykład użycia funkcji boxplot dla sumy opadów atmosferycznych na kilku stacjach

- `barplot()`

```
tidyr::spread(dane2, rok, suma) %>% as.numeric() %>% .[-1] %>% barplot(., col="lightblue")
```



Rysunek 11.3: Przykład użycia funkcji `barplot` dla sumy opadów atmosferycznych

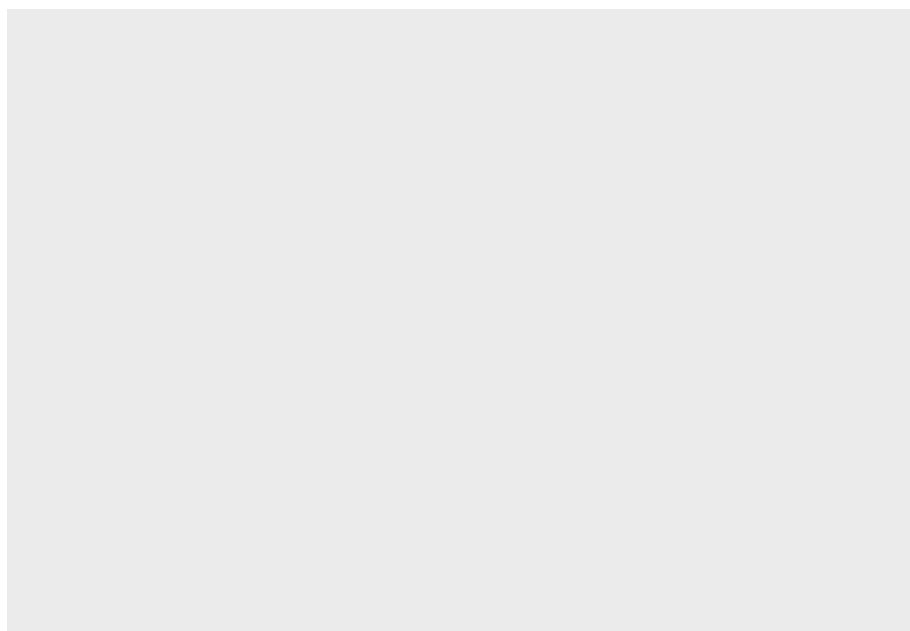
- `image()`
- `persp()`
- `wireframe()`

11.2 ggplot2

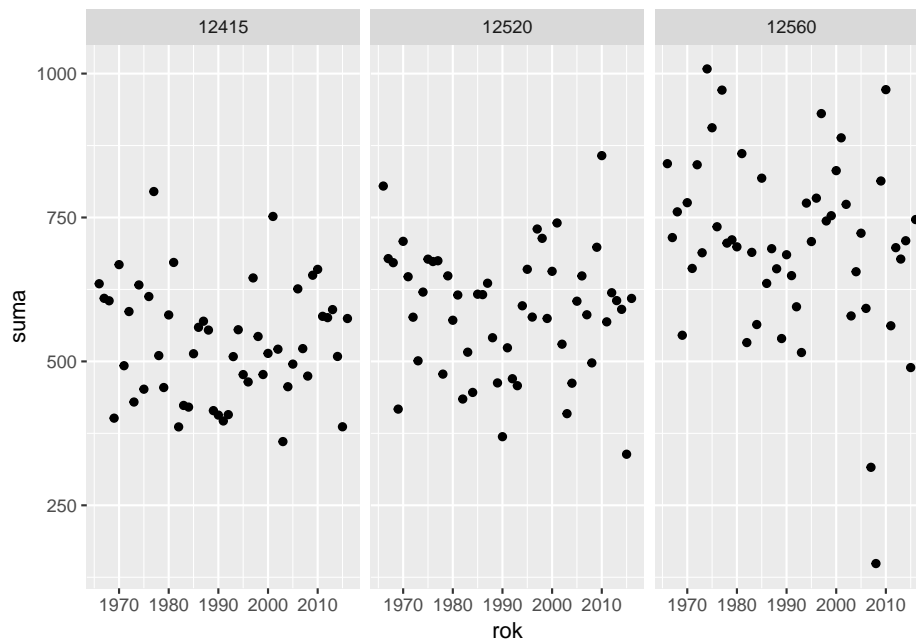
Grammar of graphics: <https://github.com/tidyverse/ggplot2/wiki>

Składnia zdecydowanie trudniejsza niż bazowego interfejsu `graphics`, jednak dająca w bardziej skrótowej postaci większe możliwości:

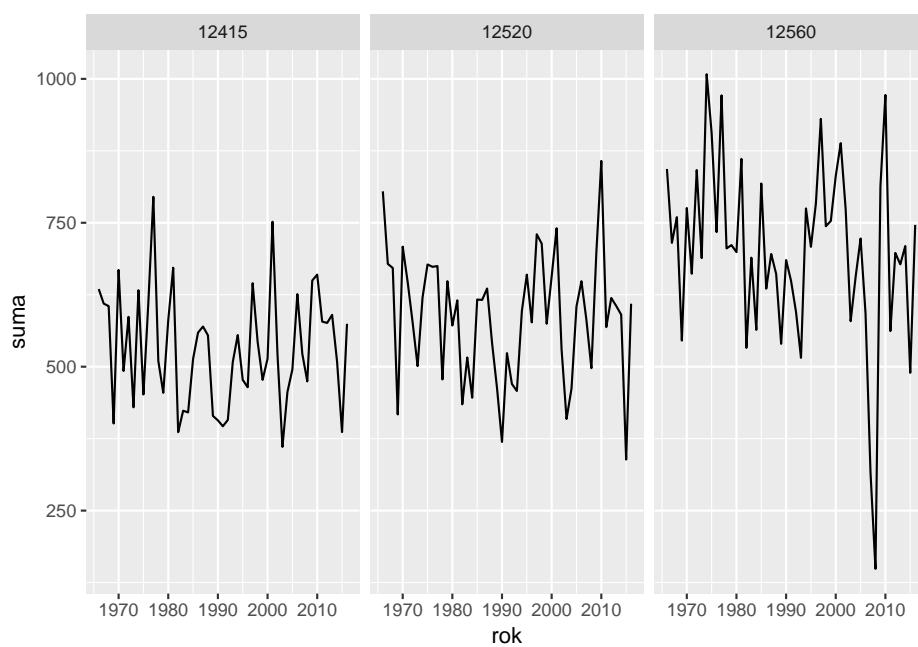
```
library(ggplot2)
ggplot(dane) #nic sie nie dzieje
```



```
ggplot(dane)+geom_point(aes(rok,suma))+facet_wrap(~stacja)
```

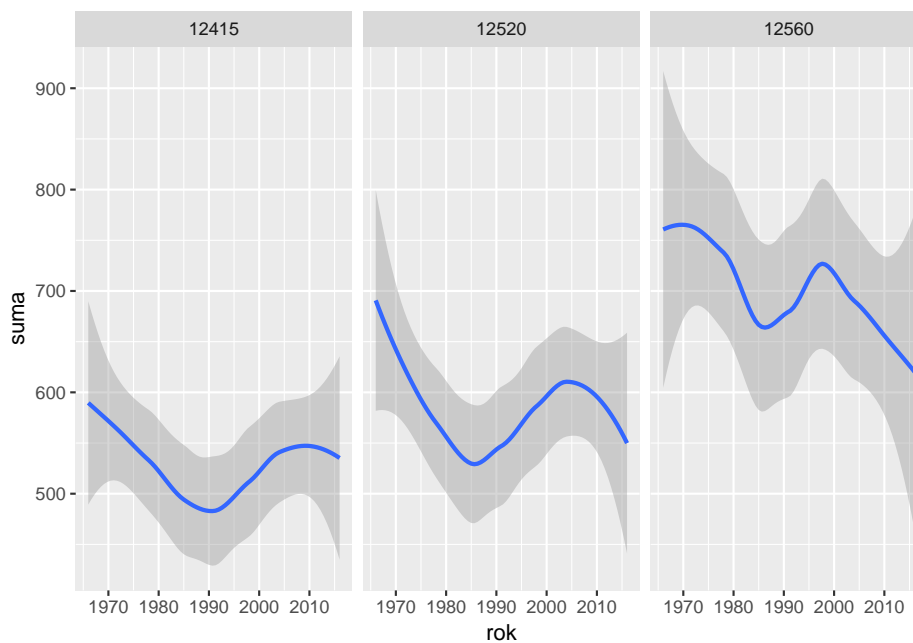


```
ggplot(dane)+geom_line(aes(rok,suma))+facet_wrap(~stacja)
```



```
ggplot(dane)+geom_smooth(aes(rok,suma),stat="smooth")+facet_wrap(~stacja)
```

```
## `geom_smooth()` using method = 'loess'
```



tutorial wprowadzający do ggplot2: <http://r-statistics.co/Complete-Ggplot2-Tutorial-Part1-With-R-Code.html>

przykłady dobrych wykresów wraz z kodem do ggplot2: <http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html>

Instalacja

Najczęściej spotykana konfiguracja środowiska pracy z **R** wymaga zainstalowania 2 programów: **R** oraz **RStudio**. Ważne, aby były one instalowane w odpowiedniej kolejności (tj. 1. **R**, 2. **RStudio**)

Instalacja R

R jest dostępny praktycznie dla każdego współczesnego systemu operacyjnego. Pliki źródłowe **R** są udostępniane na stronie internetowej CRAN (The Comprehensive R Archive Network) pod adresem <http://cran.r-project.org/>.

W przypadku większości komputerów stacjonarnych należy wybrać 64-bitową opcję instalacji i dalej postępować zgodnie ze wskazówkami instalatora.

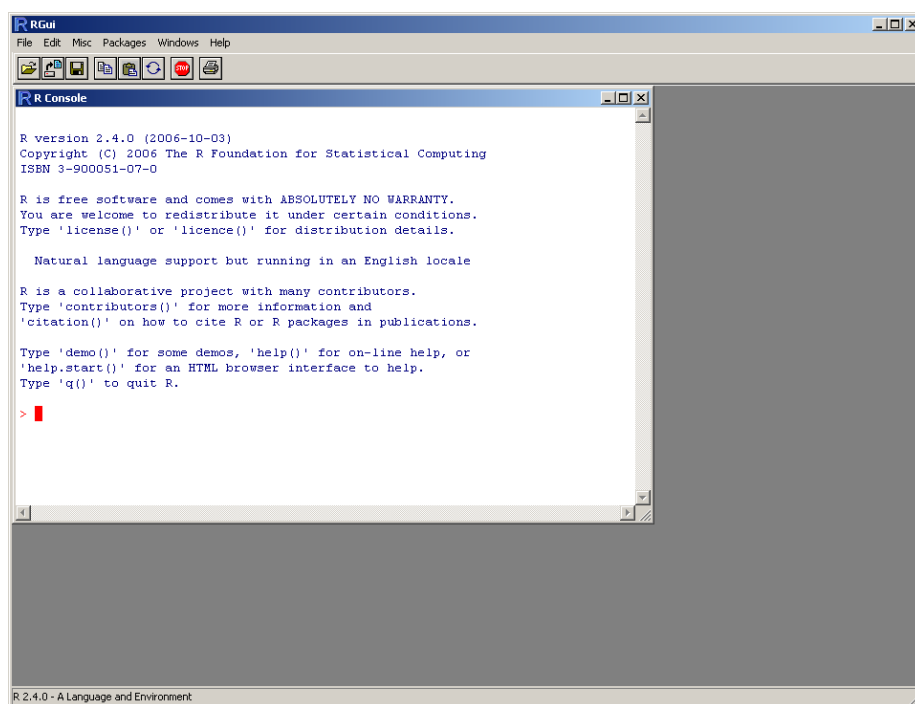
Po uruchomieniu programu okno pracy powinno wyglądać w środowisku Windows jak na poniższym obrazie:

Linux:

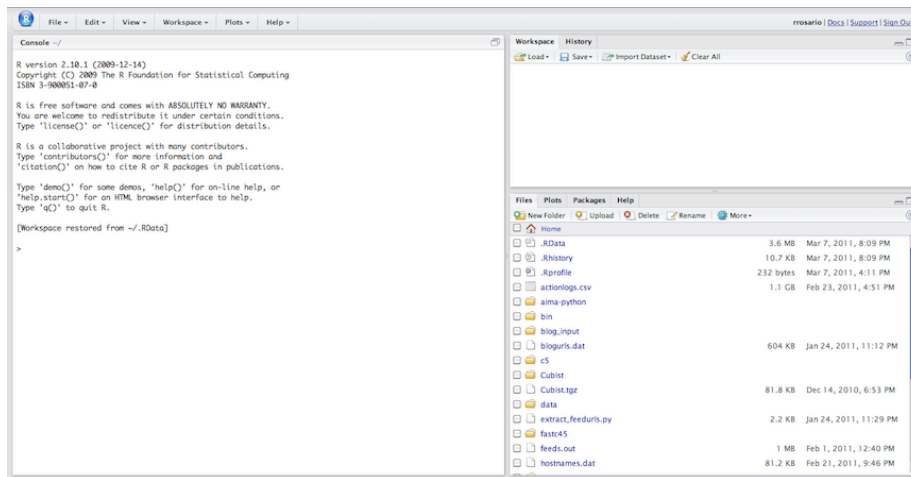
Dla systemów operacyjnych Linux zdecydowanie wygodniej wykorzystać wewnętrzne narzędzie instalacyjne. Przykładowo, dla systemu Debian/Ubuntu środowisko programistyczne **R** można zainstalować wpisując w terminalu komendę *sudo apt-get install r-base*

Instalacja RStudio

RStudio

Rysunek 11.4: Ekran początkowy programu **R** w Windowsie

Praca z **R** jest zdecydowanie bardziej przyjemna przy użyciu zintegrowanego środowiska programistycznego (IDE, ang. *Integrated Development Environment*) **RStudio**. Program ten jest swojego rodzaju nakładką graficzną na “surowy **R**” i można go pobrać ze strony <http://www.rstudio.com/products/rstudio/download/>. Po wejściu na stronę należy wybrać bezpłatną wersję programu (RStudio Desktop) odpowiednią dla naszego systemu operacyjnego z listy “*Installers for Supported Platforms*”. Po pobraniu wskazanego pliku instalator przeprowadzi nas przez intuicyjny proces instalacji. Przy pierwszym uruchomieniu program **RStudio** powinien wyglądać jak na zrzucie ekranu poniżej:



Rysunek 11.5: Ekran początkowy programu **RStudio**

RStudio dostarczane jest w wielu różnych wersjach, w tym także w bezpłatnej wersji **RStudio Server** umożliwiającej pracę zdalną z poziomu przeglądarki internetowej. Pozostałe wersje o szerszym wachlarzu zaawansowanych opcji programistycznych są udostępniane odpłatnie.

Instalacja bibliotek

R posiada ogromną liczbę bibliotek programistycznych rozszerzających jego możliwości. W niniejszym podręczniku będziemy korzystać przynajmniej z kilku takich pakietów, które przed uruchomieniem muszą zostać zainstalowane

za pomocą funkcji `install.packages()`. Przykładowo, jeśli chcesz zainstalować bibliotekę `dplyr` i masz dostęp do internetu wystarczy użyć komendy `install.packages('dplyr')`. Bibliotekę instaluje się tylko raz (chyba, że chcesz ją np. zaktualizować).

Jeśli proces instalacji przebiegł prawidłowo do aktywacji biblioteki wymagane jest użycie funkcji `library()`. W naszym przykładzie aktywacja paczki `dplyr` wymaga zastosowania polecenia `library(dplyr)` lub `library('dplyr')`. Po aktywacji paczki możemy już korzystać z nowych funkcji rozszerzających możliwości **R**.

Najczęściej kod aktywujący dane biblioteki umieszcza się na początku skryptu

Sprawy organizacyjne

->

Kolokwium końcowe - przykładowe zagadnienia:

- Wpisz komendę, która utworzy wektor 'A' składający się z wartości narastających od 1 do 10 z interwałem co 1
- Wpisz komendę, która utworzy wektor 'B' składający się z wartości -2.45, 1, 8, 9.5
- Wpisz komendę, która utworzy wektor 'C' zawierający w kolejności alfabetycznej pierwsze 3 oraz ostatnie 2 litery alfabetu łacińskiego/angielskiego
- Wpisz komendę, która utworzy wektor 'D' zawierający 200 liczb od 0 do 100 w jednakowych interwałach
- Wpisz komendę, która utworzy wektor 'E' zawierający liczby: 20.5, 19.5, 18.5, 17.5, 16.5, 15.5, 14.5, 13.5, 12.5, 11.5, 10.5, 9.5, 8.5, 7.5, 6.5, 5.5, 4.5, 3.5, 2.5, 1.5, 0.5, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 99
- Wpisz komendę, która zamieni 3-ci element wektora 'E' brakiem danych (NA)
- Wpisz komendę, która z wektora `letters` usunie: 5-ty, 10-ty i 15-ty element
- Wpisz komendę, która pozwoli na wygenerowanie 5000 losowych wartości o rozkładzie normalnym (`rnorm()`) / losowym (`runif()`)

- Wpisz komendę, która utworzy wektor składający się z 1000 elementów i każdy z nich będzie równy 5
- Wpisz komendę, która pozwoli na obliczenie: (1) sumy, (2) średniej arytmetycznej, (3) maksimum, (4) minimum, (5) odchylenia standardowego, (6) logarytmu, (7) pierwiastka kwadratowego, (8) liczby elementów (długości wektora), (9) ...

z wektora wartości `Nile` (lub wyniku polecenia `as.numeric(Nile)`)

- Jakiego argumentu należało by użyć dla podanych powyżej funkcji arytmetycznych jeśli w wektorze znajdują się braki danych (`NA`)?
- Jaka komenda pozwoli na wyświetlenie pomocy dla danej funkcji?
- Jakie zadanie wykonują funkcje: `round()`, `ceiling()`, `floor()`, `rep()`, `sort()`, `rev()`?
- Wpisz komendę, która posortuje/odwróci kolejność wektora `A` i nadpisze jego wcześniejszą zawartość
- Wpisz komendę, która stworzy wektor logiczny zawierający 2 razy wartości 'prawda' i 2 razy wartości 'fałsz'
- Stosując indeksowanie przez negację wyświetl wszystkie elementy wektora `A` bez pierwszych dwóch / bez pierwszego i ostatniego
- Stosując wyrażenia logiczne sprawdź, które elementy wektora `x` są większe od 0. Wynikiem powinien być wektor wartości logicznych (`TRUE/FALSE`)
- Napisz komendę, która zwróci indeksy elementów wektora `x`, które są mniejsze bądź równe 0. Wynikiem powinien być wektor wartości całkowitych
- Korzystając z nawiasów kwadratowych wyświetl tylko parzyste elementy wektora `letters`
- Wpisz komendę, która wyświetli zawartość kolumny `Temp` z ramki danych `airquality`
- Oblicz minimalną i/lub maksymalną wartość kolumny `Temp` z ramki danych `airquality`
- Wyświetl pierwszy rząd / pierwsze 5 rzędów ramki danych `airquality`
- Wyświetl 1, 5 i 20 rząd ramki danych `airquality`
- Oblicz średnią z pierwszej kolumny ramki danych `airquality`
- Oblicz wartość minimalną z drugiej kolumny ramki danych `airquality`
- Wyświetl pierwsze dwie kolumny ramki danych `airquality`

- Pomijając dwa pierwsze wiersze ramki danych `airquality` wyświetl wartość drugiej i trzeciej kolumny
- Stwórz dowolną ramkę danych zawierającą 3 dowolnie nazwane kolumny. Każdy wiersz powinien zawierać 2 wartości.
- Dodaj nową kolumnę o nazwie `cisnienie` do ramki danych `airquality`. Całą kolumnę wypełnij brakiem wartości (NA)
- Dodaj nową kolumnę o nazwie `TempC` do ramki danych `airquality`. Kolumnę wypełnij wynikiem działania `(airquality$Temp-32)*(5/9)`
- Wczytaj do środowiska R plik dostępny pod adresem http://biecek.pl/MOOC/dane/koty_ptaki.csv. Jego zawartość zapisz do zmiennej `dane`
- Wczytaj do środowiska R plik binarny `.Rdata`, który wcześniej pobierzesz z adresu: <http://www.enwo.pl/przetwarzanie/dane/pm10.Rdata> ; Pamiętaj o konieczności zdefiniowania wcześniej katalogu roboczego
- Wczytaj plik w formacie RDS i przypisz jego zawartość do obiektu `dane`. Dane do wczytania dostępne są pod adresem: http://www.enwo.pl/przetwarzanie/dane/pm10_new.rds
- Wpisz komendę, która zapisze obiekt `dane` do pliku RDS pod nazwą `dane.rds`
- Wpisz rozwiązanie, które złączy ciągi tekstowe zawarte w wektorach: A, B i C do postaci jednoelementowego wektora, bez spacji.
- Napisz pętlę programistyczną “for” z wbudowaną zmienną “i”, która będzie przyjmować narastające wartości liczb całkowitych od 1 do 5. Wewnątrz pętli użyj funkcji “print()”, która będzie wyświetlać chwilową wartość zmiennej “i”. Napisz kod w RStudio i przeklej go w pole odpowiedzi
- Napisz pętlę programistyczną “while” ze zdefiniowaną wcześniej zmienną “i” równą 5, która będzie się uruchamiać dla $i > 0$. Pętla powinna być konstruowana w taki sposób, aby wartość parametru “i” zmniejszała się przy każdym jej uruchomieniu o 1.
- Napisz pętlę programistyczną `for`, która będzie wyświetlać nazwy wszystkich plików znajdujących się w aktualnym katalogu roboczym
- Przekształć ciąg znaków “2015-02-22” do obiektu klasy `Date`
- Utwórz wektor klasy `Date` dla wszystkich dni w roku 2017
- Utwórz wektor klasy `Date` od 1. stycznia 2017 roku do 31. grudnia 2017 roku z interwałem co 7 dni

- Oblicz liczbę dni pomiędzy datą 10. maja 2009 roku a dzisiaj
- Przekonwertuj ciąg tekstowy “2015-02-13 12:56:26” do obiektu klasy POSIX w czasie lokalnym
- Przekonwertuj ciąg tekstowy “2015-02-13 12:56:26” do obiektu klasy POSIX w czasie ‘UTC’
- Zakres rozdziału 8. oraz 9.1-9.3 analogiczny do pytań zawartych w teście: <https://goo.gl/forms/KzrTeavRMi3dixNp2>
- Operator przetwarzania potokowego %>% można wygenerować w środowisku **RStudio** za pomocą skrótu klawiszowego ...
- Wpisz polecenie, które ze zbioru danych `airquality` wybierze jedynie kolumny, `Temp` i `Month`
- Wpisz polecenie, które ze zbioru danych `airquality` usunie wszystkie kolumny oprócz `Temp` i `Month`
- Wpisz polecenie, które ze zbioru danych `airquality` wybierze jedynie przypadki z temperaturą powyżej 86°F
- Wpisz polecenie, które posortuje zbiór danych `airquality` według narastających wartości temperatur
- Wpisz polecenie, które posortuje zbiór danych `airquality` według malejących wartości temperatur
- Korzystając z przetwarzania potokowego i funkcji pakietu `dplyr` wybierz ze zbioru danych `airquality` jedynie dni, w których (1) temperatura powietrza przekroczyła 85°F, (2) pozostawi jedynie kolumny `Temp` i `Month`, (3) posortuje te dni w kolejności od najwyższych do najniższych koncentracji ozonu
- Analogicznie do zadań podsumowujących rozdział 9.4
- Korzystając z przetwarzania potokowego oraz funkcji `group_by` oraz `summarise` oblicz średnią miesięczną temperaturę powietrza ze zbioru `airquality`
- Korzystając z przetwarzania potokowego oraz funkcji `group_by` oraz `summarise` oblicz średnią, maksymalną i minimalną miesięczną temperaturę powietrza ze zbioru `airquality`
- Korzystając z przetwarzania potokowego oraz funkcji `group_by` oraz `summarise` oblicz średnią i sumę miesięczną temperatur powietrza oraz średnią miesięczną koncentrację ozonu (po usunięciu braków w obserwacjach) ze zbioru `airquality`. Nazwij te kolumny odpowiednio: “sred”,

“suma”, “ozon”

- Wczytaj zbiór danych: `dane <- readRDS(gzcon(url("http://enwo.pl/przetwarzanie/dane/synop.rds")))`
; Następnie korzystając z przetwarzania potokowego wybierz jedynie miesiące zimowe dla Poznania i oblicz średnią temperaturę powietrza zimy. Rozwiązaniem jest wynik zaokrąglony do 2. miejsc po przecinku
- Wczytaj zbiór danych: `dane <- readRDS(gzcon(url("http://enwo.pl/przetwarzanie/dane/synop.rds")))`
; Następnie korzystając z przetwarzania potokowego wybierz jedynie miesiące letnie dla Warszawy i oblicz średnią prędkość wiatru w poszczególnych sezonach (2000, 2001, 2002, itd.). Który z sezonów letnich był najbardziej wietrzny? Podaj rok
- Wczytaj zbiór danych: `dane <- readRDS(gzcon(url("http://enwo.pl/przetwarzanie/dane/synop.rds")))`
; Następnie korzystając z przetwarzania potokowego oblicz średnią prędkość wiatru w poszczególnych latach (2000, 2001, 2002, itd.) na każdej ze stacji. Który rok i na jakiej stacji był najmniej wietrzny?
- Przekształć do postaci szerokiej dane, które można wczytać funkcją: `dane <- readRDS(gzcon(url("http://enwo.pl/przetwarzanie/dane/opady.rds")))`
. Dane w postaci szerokiej powinny zawierać w pierwszej kolumnie datę, a w kolejnych identyfikator stacji wraz z odnotowanymi wartościami opadów
- Za pomocą funkcji `gather` z pakietu `tidyr` przekształć wynik działania z wcześniejszego punktu do postaci wąskiej (pierwotnej)

Wskazówka: zadania z rozdziałów 8-10 obejmujące rozwiązanie realnych problemów z przetwarzania danych będą punktowane wyżej od zadań teoretycznych.

W trakcie kolokwium końcowego dopuszczalne jest korzystanie z podręczników, notatek oraz komputera (w tym z dostępem do internetu, bez możliwości stosowania komunikatorów i pokrewnych rozwiązań).

Bibliografia

- Biecek, P. (2008). Przewodnik po pakiecie r.
- Biecek, P. (2016). Pogromcy danych.
- Dewhurst, S. C. and Stark, K. T. (1995). *Programming in C++*. Prentice-Hall, Inc.
- Gągolewski, M. (2016). Programowanie w języku r. analiza danych, obliczenia, symulacje.
- Kalnay, E., Kanamitsu, M., Kistler, R., Collins, W., Deaven, D., Gandin, L., Iredell, M., Saha, S., White, G., Woollen, J., et al. (1996). The ncep/ncar 40-year reanalysis project. *Bulletin of the American meteorological Society*, 77(3):437–471.
- Microsoft (2016). Excel specifications and limits.
- Nowosad, J. (2016). Geostatystyka w r.
- Team, R. C. (2016). R: A language and environment for statistical computing. vienna: R foundation for statistical computing; 2016.
- Venables, W. N., Smith, D. M., Team, R. D. C., et al. (2004). An introduction to r.
- Wickham, H. and Francois, R. (2016). *dplyr: A Grammar of Data Manipulation*. R package version 0.5.0.
- Wickham, H. and Grolemund, G. (2016). R for data science.