Programowanie w R - zagadnienia wstępne

Wprowadzenie

Dotychczas nasza nauka R często nie wykraczała zbytnio poza możliwości, które możemy osiągnać w dowolnym arkuszu kalkulacyjnym. Prawdziwa moc zaklęta w językach programowania polega na zastosowaniu instrukcji sterujących, pętli oraz funkcji programistycznych, które pozwalają na automatyzację wykonywanych operacji.

Instrukcje sterujące / warunkowe

Często zdarza się, że wykonanie fragmentu kodu, chcemy uzależnić od pewnych reguł, które muszą być spełnione aby wykonać daną porcję kodu lub w zależności od testowanego wyrażenia chcemy uruchomić inny zestaw reguł. Do tego celu najlepiej użyć instrukcji sterujących **if**, **if / else** lub **if / else / elif**.

Instrukcja `if`

Najprostszą formą jest instrukcja warunkowa **if** w postaci:

```
#Instrukcja warunkowa 'if':
 if(warunek){
     instrukcja1
     }
```

Zadanie 1:

- Przypomnij sobie działanie operatora `%%` na dowolnych dwóch liczbach całkowitych. Po lewej stronie ustaw liczbę większą, po prawej od `%%` liczbę mniejszą. Następnie zmniejszaj bądź zwiększaj liczbę po prawej stronie. Co jest wynikiem działania operatora `%%`?
- Napisz instrukcję warunkową `if` która będzie sprawdzała czy dana liczba jest podzielna w całości przez 5. Jeśli tak, wykorzystaj funkcje `print` oraz `paste` (lub `cat`), która wydrukuje na ekranie komunikat *liczba ___ jest podzielna przez 5*. W przeciwnym razie nie powinno się wyświetlić się nic.

Instrukcja `if else`

Instrukcja `if` jest często łączana do postaci instrukcji `if-else`, gdzie po przetestowaniu pierwszego warunku logicznego i otrzymaniu wyniku `FALSE` wykonywania jest instrukcja alternatywna zapisana w bloku `else` Ogólna forma tej instrukcji przyjmuje postać:

```
#Instrukcja warunkowa 'if-else':
 if (warunek){
     instrukcja1
     } else {
     instrukcja2
     }
```

Zadanie 2:

- Przypomnij sobie działanie operatora **%/%** na dowolnych dwóch liczbach całkowitych. Analogicznie jak w zadaniu 1 po lewej stronie ustaw liczbę większą. Następnie zmniejszaj bądź zwiększaj liczbę po prawej stronie. Co jest wynikiem działania operatora **%/%**?
- Napisz instrukcję warunkową `if else` która będzie sprawdzała czy dana liczba jest podzielna w całości przez 4. Jeśli tak, wykorzystaj funkcje `print` oraz `paste` (lub `cat`), która wydrukuje na ekranie komunikat *liczba X jest podzielna przez 5*. W przeciwnym razie wyświetl komunikat: *liczba ___ nie jest podzielna przez 5; Reszta z dzielenia ___ przez 5 wynosi _____*

Instrukcja `if - else if - elif`

Co w przypadku kiedy chcemy rozbudować nasze drzewo decyzyjne? Możemy wówczas przetestować więcej wyrażeń logicznych w blokach `if - else if - elif`. Dodając `elif` można dowolnie manipulować kontrolą nad wykonywanymi blokami kodu. Jeśli żadna z instrukcji w blokach `if - else if` nie jest poprawna, wóœczas wykonuje się fragment kodu zapisany w bloku `else`. Ogólna forma tej instrukcji przyjmuje postać:

```
#Instrukcja warunkowa 'if -else if - elif':
if (warunek1) {
    wyrazenie1
    } else if (warunek2) {
    wyrazenie2
    } else if  (warunek3) {
    wyrazenie3
    } else {
    wyrazenie4
}
```

Zadanie 3:

- Stwórzmy rozbudowany algorytm do przeliczania temperatury, który na podstawie użytego słowa kluczowego będzie przeliczać jednostki ze stopni Celsjusza na Kelwiny, Fahrenheity bądź pozostawiać tą wartość bez zmian jeśli wybrano Celsjusze. Dodatkowo sprawdźmy czy podana wartość jest typu numerycznego.
- Do obiektu `temp` przypisz dowolną wartość temperatury powietrza
- Do obiektu `konwersja` przypisz wartość **“ce”**, **“ke”** lub **“fa”** w zależności od tego w którą stronę ma być przekonwertowana temperatura
- Napisz instrukcję warunkową `if - else if - elif` która w bloku `if` będzie sprawdzała za pomocą komendy `is.numeric()` czy zawartość obiektu `temp` jest typu numerycznego. Jeśli jednocześnie obiekt `konwersja == tc` wyświetl komunikat: **Temperatura `temp` to zmienna typu numerycznego i wynosi _____**
- W pierwszym bloku `else if` sprawdź czy zmienna `konwersja` ma wartość **fa** i jeśli tak wyświetl komunikat: **Temperatura `temp` C to _____ F**
- W drugim bloku `else if` sprawdź czy zmienna `konwersja` ma wartość **ke** i jeśli tak wyświetl komunikat: **Temperatura `temp` C to _____ K**
- Jeśli żaden z warunków nie jest spełniony wyświetl informację o tym, że zmienna `konwersja` nie ma wymaganej wartości i powinna ona być równa **“ke”** , **“fa”** lub **“ce”**

Zadanie 3.1 (opcjonalne):

- Stwórz grę w której przeciwnik musi zgadnąć na jakiej wysokości leci pilot (np. zmienna `wysokosc` w przedziale 0 - 10000).
- Jeśli użytkownik zgadnie - wyświetl komunikat
- Jeśli użytkownik nie zgadnie i wynik przeszacował - wyświetl komunikat "za wysoko"
- Jeśli użytkownik nie zgadnie i niedoszacował - wyświetl komunikat "za nisko"

Instrukcja `ifelse`

Niejednokrotnie zdarza się, że potrzebne jest wykorzystanie analogu dla funkcji **jeżeli** znanej z arkuszy kalkulacyjnych, która umożliwia szybką konwersję wartości w zależności od testowanego warunku logicznego. Przykładowo, pewne funkcje mogą zwrócić wynik wykraczający poza możliwy fizycznie zakres i chcemy je przekonwertować do innej (np. maksymalnej) możliwej wartości. Funkcja `ifelse` umożliwia przetestowanie nie tylko pojedynczej, ale całego wektora wartości biorąc pod uwagę czy spełnia dany warunek logiczny. Ogólna postać `ifelse`:

```
#Instrukcja warunkowa 'ifelse':
ifelse(warunek, wartosc_dla_TRUE, wartosc_dla_false)
```

Zadanie 4:

- Wygeneruj zbiór 30 wartości losowych z rozkładu normalnego komendą `rnorm(30)` i przypisz jej rezultat do obiektu `a`
- Następnie wartości poniżej 0 zamień na 0 a wartości dodatnie pozostaw bez zmian. Wynik działania ponownie przypisz do obiektu `a` i wyświetl jego aktualną wartość

Pętla programistyczna `for`

Często zdarza się, że chcemy wykonać jakąś operację więcej niż jeden raz, zmieniając tylko fragmentarycznie zakres naszego postępowania. Przykładowo, możemy chcieć wykonać oddzielne wykresy dla każdej kolumny, rzędu, wczytać i policzyć ustalone statystyki dla wszystkich plików znajdujących się w naszym katalogu roboczym, itp. W tym celu przydatna może okazać się pętla `for` w ogólnej postaci:

```
#Pętla programistyczna for:
for(zmienna_tymczasowa in zakres_wartosci){
  wykonaj_jakis_fragment_kodu
}
```

W powyższym fragmencie `zmienna_tymczasowa` za każdym razem przyjmuje kolejną (pojedynczą) wartość z wektora `zakres_wartosci`. Obiekt `zmienna_tymczasowa` nie powinien być wcześniej zdefiniowany (tj. jest definiowany "w locie" w trakcie kolejnych iteracji pętli). Pętla wykonuje się tak długo dopóki nie pojawi się błąd lub do momentu użycia wszystkich elementów z wektora `zakres_wartosci`.

Zadanie 5a:

- Ile razy wykona się poniższa pętla?

```
#Pętla programistyczna for:
for(i in -1:5){
  print(i)
}
```

Zadanie 5b:

Pobierz współrzędne stacji meteorologicznych w 5 dowolnych krajach Europy za pomocą paczki `climate`:

- Zainstaluj i aktywuj paczkę `climate`
- Następnie stwórz wektor `kraje` zawierający nazwy 5 dowolnych europejskich krajów (nazwy powinny być zapisane w j. angielskim)
- Stwórz pusty obiekt `wynik`, w którym będziemy przechowywać metadane stacji w wybranych 5 krajach
- Rozpocznij pętlę **for** w której będziemy iterować po zmiennej `kraje`
- Zapoznaj się z instrukcją dla funkcji `stations_ogimet`. Wewnątrz pętli wykorzystaj tą funkcję do zapisania współrzędnych stacji do obiektu `tmp`
- Opcjonalnie - przy każdej iteracji dodaj do tymczasowej ramki danych `tmp` kolumnę z nazwą kraju
- Złącz obiekt `tmp` z obiektem `wynik` np. za pomocą funkcji `rbind()` lub `rbind.data.frame()`
- Po opuszczeniu pętli wyświetl stworzony obiekt
- Wygeneruj roboczą mapę z pobranymi danymi (np. za pomocą funkcji `plot()`)
- Który z wybranych krajów miał najwięcej stacji? (np. za pomocą funkcji `table()`)

```
##   wmo_id station_names      lon      lat alt    kraj
## 1  12756     Szecseny  19.51668 48.11667 152 Hungary
## 2  12766      Josvafo  20.53334 48.48334 305 Hungary
## 3  12772      Miskolc  20.76668 48.08333 232 Hungary
## 4  12786       Zahony  22.16667 48.40001 103 Hungary
## 5  12805       Sopron  16.60001 47.68335 233 Hungary
## 6  12812  Szombathely  16.63335 47.26667 220 Hungary
## [1] "..."
##   wmo_id     station_names      lon      lat alt   kraj
## 1  12001 Petrobaltic Beta  18.16667 55.46668  46 Poland
## 2  12100        Kolobrzeg  15.58334 54.18334   3 Poland
## 3  12105         Koszalin  16.15000 54.20000  32 Poland
## 4  12115            Ustka  16.86668 54.58335   6 Poland
## 5  12120             Leba  17.53334 54.75001   2 Poland
## 6  12124         Darlowek  16.40001 54.40001   2 Poland
## [1] "..."
##   wmo_id  station_names      lon      lat  alt  kraj
## 1  11406          Cheb  12.40001 50.08333  470 Czech
## 2  11414  Karlovy Vary  12.91668 50.20000  603 Czech
## 3  11423        Primda  12.66668 49.66668  742 Czech
## 4  11438      Tusimice  13.33334 50.38334  322 Czech
## 5  11450 Plzen-Mikulka  13.38334 49.76668  360 Czech
## 6  11457      Churanov  13.61668 49.06667 1122 Czech
## [1] "..."
##   wmo_id       station_names      lon      lat alt     kraj
## 1  11801            Malacky  17.15000 48.40001 205 Slovakia
## 2  11812      Maly Javornik  17.15000 48.25001 585 Slovakia
## 3  11813  Bratislava-koliba  17.11667 48.16667 288 Slovakia
## 4  11816  Bratislava Ivanka  17.20000 48.20000 133 Slovakia
## 5  11819 Jaslovske Bohunice  17.66668 48.48334 176 Slovakia
## 6  11826           Piestany  17.83335 48.61668 163 Slovakia
## [1] "..."
```

Pętla programistyczna `while`

W zasadzie niemal wszystkie kwestie związane z działaniem na pętlach można rozpisać przy pomocy pętli `for`. Niemniej jednak w sporadycznych przypadkach warto wykorzystać pętlę `while`, która nie ma z góry określonej liczby iteracji, a jej wykonywanie jest uzależnione od testowanego warunku logicznego. Jeśli po słowie kluczowym `while()` testowane w nawiasie wyrażenie logiczne zwraca `TRUE` wówczas wykonywana jest instrukcja wewnątrz pętli. Ważne jest, aby wyrażenie testowane po słówku `while()` działało na istniejących obiektach, stąd też często obiekty które są użyte do testu logicznego są tworzone bezpośrednio przed samą pętlą. Ogólna postać pętli `while` wygląda następująco:

```
#Pętla programistyczna while:
while (testowane_wyrazenie){
   instrukcje_do_wykonania
}
```

Zadanie 6a:

- Opisz schemat działania poniższej pętli `while`:

```
#Przykładowa pętla programistyczna while:
losowanie = 0
while(losowanie < 0.9) {
   losowanie = runif(1)
   print(losowanie)
}
```

```
## [1] 0.2655087
## [1] 0.3721239
## [1] 0.5728534
## [1] 0.9082078
```

Zadanie 6b:

Stwórz "program" do losowania totolotka bazujący na funkcji `sample()` i pętli programistycznej `while`

- Stwórz wektor `mojlos` z 6 wartościami (ulubionych) liczb z przedziału od 1 do 49
- Stwórz pusty obiekt `wynik` w którym będziesz przechowywać liczbę trafień pojedynczego losowania totolotka
- Stwórz zmienną `licznik` bezpośrednio przed rozpoczęcie pętli `while` w której będziesz zliczać liczbę wykonywanych iteracji.
- Rozpoczynając pętle `while` umieść w nawiasie test logiczny, który pozwoli uruchamiać "maszynę losującą" tak długo dopóki zmienna `licznik` nie przekroczy wartości 50
- Wewnątrz pętli `while` użyj komendy `sample()`, która wykona losowanie 6 z 49 liczb i zapisze wynik losowania w postaci nowego obiektu `losowanie`
- Zlicz za pomocą operatora `%in%` ile razy liczby z obiektu `losowanie` pokrywają się z zawartością obiektu `mojlos`. Użyj w tym celu operatora `%in%`. Wynik ile razy trafiona została dane kombinacja (tj. od 0 do 6) dopisz do istniejącego wektora `wynik`
- Pod koniec pętli nie zapomnij o zwiększeniu obiektu `licznik` o 1 w celu uniknięcia działania pętli w nieskończoność
- Wyświetl aktualny numer iteracji za pomocą funkcji `print()` lub `cat()` + `paste()`
- Po uruchomieniu pętli sprawdź ile razy "wygrałeś". Przydatna może okazać się funkcja `table()`

Zadanie 6c:

- Opcjonalnie - przepisz powyższy kod pętli `while` w sposób który będzie działać do momentu wylosowania (przynajmniej) "trójki".

Funkcje

Kolejną rzeczą, którą każdy początkujący programista **R** musi poznać są funkcje. Wszystkie polecenia, po których do tej pory wywoływaliśmy `()` są w rzeczywistości właśnie funkcjami przyjmującymi pewne parametry. Jeśli wywołamy jakieś polecenie będące funkcją bez nawiasów, wówczas wyświetli nam się jej kod źródłowy. Wpisz np. polecenie `log` bez nawiasów aby zobaczyć jej wynik.

Ogólna postać definiowania nowej (własnej) funkcji wygląda następująco:

```
#Funkcje:
nazwafunkcji  =  function(argument1, argument2, ...) {
    wyrażenia zawarte w funkcji
  return(nazwa_zwracanego_obiektu)
  # jeśli brak komendy return wówczas zwracany jest ostatni obiekt funkcji

} # koniec definiowania funkcji
```

Następnie wywołujemy naszą własną funkcję wpisując jej nazwę i argumenty.

Spróbujmy stworzyć własną funkcję, której zadaniem będzie konwersja temperatury ze stopni Celsjusza na Fahrenheity. Nazwij funkcję `tc2tf` i przetestuj jej wynik na dowolnej liczbie (np. 20):

```
tc2tf  =  function(tc) {
    tf = tc * 1.8 + 32
    return(tf)
}

tc2tf(20)
```

```
## [1] 68
```

Często zdarza się, że dla pewnych argumentów funkcji definiowane są wartości domyślne (jeśli np. są stosowane często i nie chcemy ich za każdym razem podawać):

```
tc2tf  =  function(tc = 30) {
    tf = tc * 1.8 + 32
    return(tf)
}

tc2tf(20) # definiujemy argument
```

```
## [1] 68
```

```
tc2tf(tc = 25) # podajemy dokładnie który argument chcemy zdefiniować
```

```
## [1] 77
```

```
tc2tf() # wykorzystujemy argument domyślny
```

```
## [1] 86
```

Zadanie 7a - Wzór barometryczny (niwelacja barometryczna)

Napisz funkcję, która pozwoli na obliczenie różnicy wysokości na której zmierzono 2 wartości ciśnień.

Można do tego celu zastosować przekształconą postać wzoru barometrycznego (https://pl.wikipedia.org/wiki/Wz%C3%B3r_barometryczny)

$h = -\frac{RT}{gu} * ln\frac{p}{p0}$

gdzie:

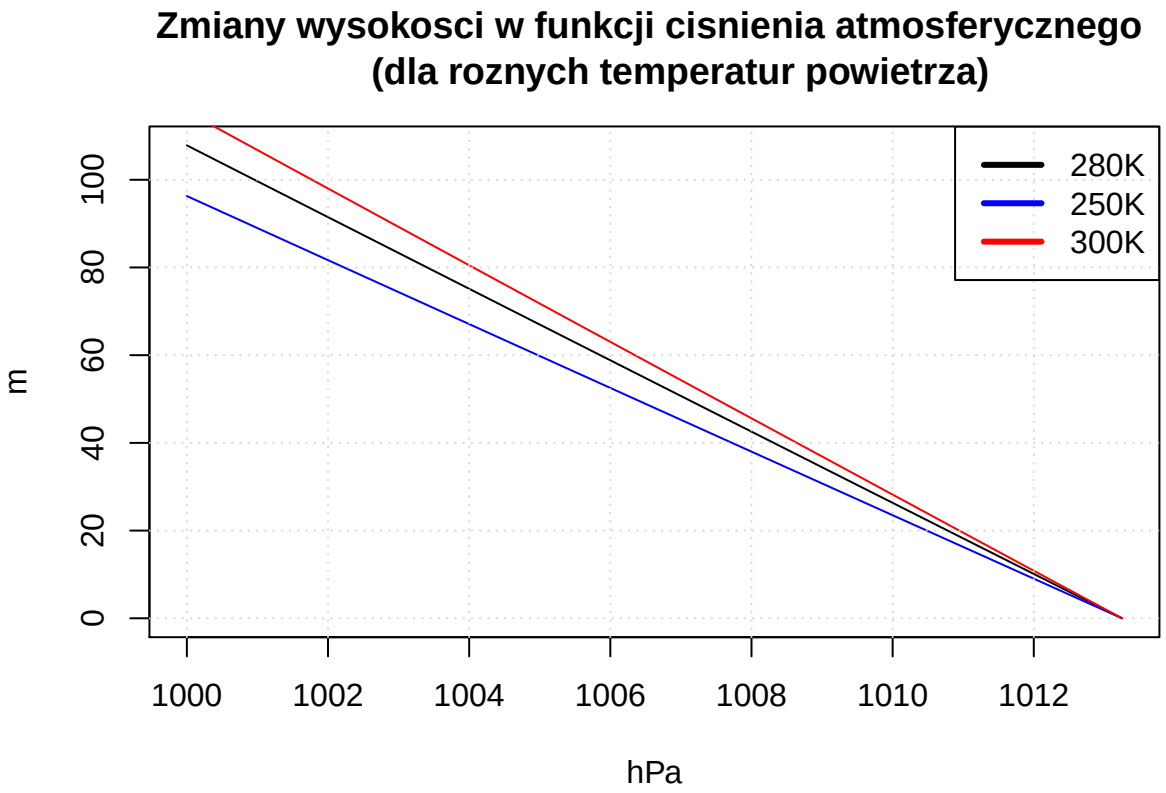
```
p0 - ciśnienie atmosferyczne na poziomie odniesienia,
u - masa molowa powietrza (0.0289644 kg/mol),
g - przyspieszenie ziemskie,
R - stała gazowa (8.314472 J/(mol.K) )
T - temperatura powietrza w K.
```

Nastepnie oszacuj jaka różnica ciśnień wystąpi pomiędzy parterem a 12 piętrem wieżowca podstawiając różne wartości dla stworzonej funkcji:

```
babinet = function(p, p0 = 101325, t = 280){
  u = 0.0289644
  R = 8.314472
  g = 9.81
  h = -(R*t)/(g*u) * log(p/p0)
  return(h)
}
```

Użyjmy funkcję do stworzenia wykresu:

```
plot(y = babinet(p = 101325:100000),
     x = 101325:100000/100, type = 'l', xlab = 'hPa', ylab = 'm',
     main = "Zmiany wysokosci w funkcji cisnienia atmosferycznego
     (dla roznych temperatur powietrza)")
lines(y = babinet(p = 101325:100000, t = 250), x = 101325:100000/100, col = 'blue')
lines(y = babinet(p = 101325:100000, t = 300), x = 101325:100000/100, col = 'red')
grid()
legend("topright", col = c("black", "blue","red"),
       legend = c("280K", "250K", "300K"), lty=1, lwd=3)
```



Zadanie 7b

- Zmodyfikuj funkcję `babinet()` do postaci wzoru barometrycznego, która pozwoli na obliczenie wartości ciśnienia na zadanej wysokości.

Wykorzystaj poniższy wzór:

$p = p0 * exp(\frac{-ugh}{RT})$

- Narysuj wykres zmian ciśnienia w przekroju od poziomu morza do wybranej wysokości (np. Mt. Blanc lub Mt. Everest)

Kiedy stosowac funkcje?

Dobrym zwyczajem jest "owijanie" często używanych bloków kodu do postaci funkcji. Jeśli korzystamy z jakiegoś schematu postępowania bardzo często (np. rysowania spersonalizowanych wykresów) warto zawrzeć je właśnie w postaci funkcji z własnymi argumentami.

Funkcje niekiedy stają się koniecznością. Dotyczy to zwłaszcza efektywnego działania na listach...

Listy...