Computerphysik Programmiertutorial 12

Prof. Dr. Matteo Rizzi und Dr. Markus Schmitt - Institut für Theoretische Physik, Universität zu Köln

ILIAS: https://www.ilias.uni-koeln.de/ilias/goto_uk_crs_3862489.html

Github: https://github.com/markusschmitt/compphys2021

Inhalt dieses Notebooks: Dateien lesen und schreiben "zu Fuß", Dateien lesen und schreiben mit `JLD`, `do`-Block Syntax, Pipelines von Funktionen, Unicode Zeichen

In [ ]:
```
# using Pkg; Pkg.add("JLD")
```

Dateien lesen und schreiben "zu Fuß"

Bisher haben wir gesehen wie man `DataFrame`s aus `.csv`-Dateien auslesen oder in `.csv`-Dateien schreiben kann.

Julia bietet auch die Möglichkeit Daten unstrukturiert in Dateien zu schreiben.

Die Funktion `write(<Dateiname>, <Variable>)` schreibt den Wert von `<Variable>` in die Datei `<Dateiname>`.

In [1]:
```
write("meine_datei.txt", "Hallo!")
```

Out[1]:
```
6
```

Falls die Datei bereits existiert, wird sie überschrieben:

In [2]:
```
write("meine_datei.txt", "Hallo nochmal!")
```

Out[2]:
```
14
```

Es können auch mehrere Variablen hintereinander geschrieben werden:

In [3]:
```
write("meine_datei.txt", "Hallo nochmal!", "13")
```

Out[3]:
```
16
```

Der Rückgabewert von `write()` gibt an wie viele Bytes geschrieben wurden.

Dateien explizit öffnen

Die Funktion `open(<Dateiname>, <(Schreib-/Lese-)modus>)` öffnet eine Datei und gibt einen `IOStream` zurück. Ein `IOStream` ist eine Datenstruktur, die die geöffnete Datei "verwaltet" und uns ermöglicht in der Datei zu schreiben/lesen.

Bei Öffnen können wir den Schreib-/Lesemodus wählen. Optionen sind:

- `"w"`: Schreiben und Datei überschreiben, falls sie bereits existiert.
- `"a"`: Schreiben und an Datei anhängen, falls sie bereits existiert.
- `"r"`: Lesen.

In [4]:
```
io = open("meine_datei.txt", "w")
write(io, "Hallo!")
close(io)
```

Überschreiben:

In [5]:
```
io = open("meine_datei.txt", "w")
write(io, "Nochmal hallo!")
close(io)
```

Anhängen:

In [6]:
```
io = open("meine_datei.txt", "a")
write(io, " Ich habe noch mehr mitzuteilen.")
close(io)
```

In die geöffnete Datei können wir immer weitere Daten schreiben, indem wir `write()` wiederholt aufrufen. Hier schreiben wir eine Reihe von Zufallszahlen in die Datei, jede gefolgt von dem Zeichen `\n`, das ein Zeilenende markiert.

In [7]:
```
io = open("meine_datei.txt", "w")

for i in 1:10
    write(io, "$(rand())\n")
end

close(io)
```

Dateien auslesen

Der Inhalt einer Datei kann mit `read(<Dateiname>, <Datentyp>)` ausgelesen werden. Dabei muss neben dem Dateinamen der `<Datentyp>` angegeben werden. Die Datei liegt auf der Festplatte als eine Folge von Bits, die Julia nur sinnvoll interpretieren kann, wenn wir vorgeben um welchen Datentyp es sich handelt.

In [8]:
```
read("meine_datei.txt", String)
```

Out[8]:
```
"0.7897354289037766\n0.2575976421889046\n0.4632817452923046\n0.7423763697216876\n0.35556570791607234\n0.10927117716771306\n0.8693070269663374\n0.63324454816245
29\n0.09590088642199213\n0.9037596320233299\n"
```

Die Funktion `readline(<IOStream>)` liest die Datei nur bis zum nächsten Zeilenende, das mit `\n` markiert ist

In [9]:
```
io = open("meine_datei.txt", "r")

readline(io)
```

Out[9]:
```
"0.7897354289037766"
```

Wenn wir eine Datei z.B. mit `readline` nur Stückweise auslesen, "merkt sich" die `IOStream` Datenstruktur die Stelle, bis zu der wir gelesen haben (mit einem "file pointer"). Beim nächsten Aufruf wird dann von dort weitergelesen:

In [10]:
```
readline(io)
```

Out[10]:
```
"0.2575976421889046"
```

In [11]:
```
close(io)
```

Die Funktion `eof(<IOStream>)` testet ob beim Auslesen das Ende einer Datei erreicht ist. Das können wir nutzen um eine Datei vollständig zeilenweise auszulesen:

In [12]:
```
io = open("meine_datei.txt", "r")

while !eof(io)
    println("nächste Zeile:")
    println(readline(io))
end

close(io)
```

```
nächste Zeile:
0.7897354289037766
nächste Zeile:
0.2575976421889046
nächste Zeile:
0.4632817452923046
nächste Zeile:
0.7423763697216876
nächste Zeile:
0.35556570791607234
nächste Zeile:
0.10927117716771306
nächste Zeile:
0.8693070269663374
nächste Zeile:
0.6332445481624529
nächste Zeile:
0.09590088642199213
nächste Zeile:
0.9037596320233299
```

Binäre Dateien

Bisher haben wir nur `String`s in Dateien geschrieben. Tatsächlich schreibt `write` die Daten einfach in Binärformat in die Datei. Daher können wir Variablen von beliebigen Datentypen auf die Festplatte schreiben:

In [15]:
```
io = open("meine_datei.txt", "w")

for i in 1:10
    r = rand()
    println(r)
    write(io, r)
end

close(io)
```

```
0.49915663672554067
0.3227471160192137
0.19638682266272345
0.21174619266013073
0.5395575467433462
0.17394113162357794
0.8974987485564483
0.043498301647241444
0.818381102785956
0.5106443896449295
```

Diese Zufallszahlen vom Typ `Float64` können wir auch der Reihe nach wieder auslesen:

In [16]:
```
io = open("meine_datei.txt", "r")

while !eof(io)
    println("nächste Zahl:")
    println(read(io, Float64))
end

close(io)
```

```
nächste Zahl:
0.49915663672554067
nächste Zahl:
0.3227471160192137
nächste Zahl:
0.19638682266272345
nächste Zahl:
0.21174619266013073
nächste Zahl:
0.5395575467433462
nächste Zahl:
0.17394113162357794
nächste Zahl:
0.8974987485564483
nächste Zahl:
0.043498301647241444
nächste Zahl:
0.818381102785956
nächste Zahl:
0.5106443896449295
```

Achtung: Beim Auslesen muss der Datentyp natürlich mit dem Datentyp übereinstimmen, der geschrieben wurde!

Beispiel: Schreibe `String`s

In [19]:
```
io = open("meine_datei.txt", "w")

for i in 1:10
    r = rand()
    write(io, "$r\n")
    println(r)
end

close(io)
```

```
0.4418222914007677
0.13609292509372306
0.5619073940749502
0.32347577544655226
0.53982787505793
0.18303003670700413
0.9316899775748613
0.9743361485736846
0.542948075557792
0.3451031106687308
```

Lese `Float64`s

In [20]:
```
io = open("meine_datei.txt", "r")

while !eof(io)
    println(read(io, Float64))
end

close(io)
```

```
6.757988440108582e-67
1.586603459105611e-47
1.3131775242310594e-47
3.121508272773327e-33
2.8296053238665905e-77
5.6367811516668945e-62
1.8328931739898532e-76
3.058114120251466e-57
1.3824094678284254e-47
3.22524432647159e-86
1.0858655092040017e-42
2.8283551108917656e-77
1.399055352701451e-76
2.5789516865746856e-57
1.0880036762041923e-71
9.51344280440617e-43
3.225235110124066e-86
2.741984991768851e-57
1.3834796014479163e-47
4.013971764196554e-57
1.0406756783735267e-42
2.108970418931774e-52
1.5181817143723214e-47
```

```
EOFError: read end of file

Stacktrace:
 [1] read(s::IOStream, T::Type{Int64})
   @ Base ./iostream.jl:410
 [2] read(s::IOStream, #unused#::Type{Float64})
   @ Base ./iostream.jl:419
 [3] top-level scope
   @ ./In[20]:4
 [4] eval
   @ ./boot.jl:360 [inlined]
 [5] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1094
```

Es können auch Arrays in Dateien geschrieben werden:

In [21]:
```
A = rand(3,4,2)
```

Out[21]:
```
3×4×2 Array{Float64, 3}:
[:, :, 1] =
 0.375028  0.917865  0.931653  0.399956
 0.443959  0.212559  0.218085  0.713353
 0.724981  0.798036  0.788469  0.719668

[:, :, 2] =
 0.574085  0.525926  0.544969  0.276844
 0.948426  0.482921  0.4347    0.0258057
 0.205407  0.189675  0.097316  0.00870856
```

In [22]:
```
io = open("meine_datei.txt", "w")
write(io, A)
close(io)
```

Zum Auslesen legen wir ein leeres Array `B` mit gleicher Größe und gleichem Datentyp an und verwenden die `read!()` Funktion:

In [23]:
```
B = Array{Float64,3}(undef,3,4,2)

io = open("meine_datei.txt", "r")
read!(io, B)
close(io)
```

In [24]:
```
isapprox(A,B)
```

Out[24]:
```
true
```

Daten lesen/schreiben mit `JLD`

Mit dem Paket `JLD` können wir beliebige Datenstrukturen sehr einfach in Dateien schreiben und auslesen (Dokumentation).

In [25]:
```
using JLD
```

Daten verschiedener Datenstrukturen speichern:

In [26]:
```
t = 15
z = Dict("a" => 1, "b" => "petra")
save("meine_andere_datei.jld", "t", t, "z", z, "array", A)
```

Gespeicherte Daten laden:

In [27]:
```
d = load("meine_andere_datei.jld")
```

Out[27]:
```
Dict{String, Any} with 3 entries:
  "array" => [0.375028 0.917865 0.931653 0.399956; 0.443959 0.212559 0.218085 0…
  "t"     => 15
  "z"     => Dict{String, Any}("b"=>"petra", "a"=>1)
```

In [28]:
```
d["t"]
```

Out[28]:
```
15
```

`do`-Block Syntax

Der `do`-Block ist eine alternative Syntax um einer Funktion als erstes argument eine anonyme Funktion zu übergeben.

Ein Beispiel ist die `map` funktion, die ihr erstes Argument (eine Funktion) elementweise auf das Array anwendet, das als zweites Argument übergeben wird:

In [29]:
```
arr = [17, -16, 0]

map(x->begin
            if x < 0 && iseven(x)
                return 0
            elseif x == 0
                return 1
            else
                return x
            end
        end,
    arr)
```

Out[29]:
```
3-element Vector{Int64}:
 17
  0
  1
```

Mit der `do`-Block Syntax kann das gleich in der folgenden Form geschieben werden:

In [30]:
```
map(arr) do x
    if x < 0 && iseven(x)
        return 0
    elseif x == 0
        return 1
    else
        return x
    end
end
```

Out[30]:
```
3-element Vector{Int64}:
 17
  0
  1
```

Die `do`-Block Syntax wird häufig zum Öffnen von Dateien verwendet, da die `open()`-Funktion die Datei dann automatisch wieder schließt:

In [31]:
```
open("meine_datei.txt", "w") do io
    for i in 1:10
        write(io, "$(rand())\n")
    end
end
```

In [32]:
```
open("meine_datei.txt", "r") do io
    while !eof(io)
        println(readline(io))
    end
end
```

```
0.3730947284243875
0.9419933259977824
0.8643311124158946
0.025852021733607122
0.5501102301622851
0.49149327795585007
0.5737324097164058
0.52600446229939
0.05708012386328276
0.608512901498754
```

Pipelines von Funktionen

Verschachtelte Funktionsaufrufe können alternativ als sogenannte *pipeline* geschrieben werden.

In [33]:
```
f(x) = 3*x + 5
g(x) = x^2.4 - x
```

Out[33]:
```
g (generic function with 1 method)
```

Verschachtelter Funktionsaufruf:

In [34]:
```
f(log(g(13)))
```

Out[34]:
```
23.383755485327974
```

Das gleiche mit dem pipeline-Operator `|>`:

In [35]:
```
g(13) |> log |> f
```

Out[35]:
```
23.383755485327974
```

Eine weitere Möglichkeit ist, den Verknüpfungsoperator `∘` zu verwenden:

In [36]:
```
(f ∘ log ∘ g)(13)
```

Out[36]:
```
23.383755485327974
```

In [37]:
```
fun = f ∘ log ∘ g
```

Out[37]:
```
f ∘ log ∘ g
```

In [38]:
```
fun(13)
```

Out[38]:
```
23.383755485327974
```

Unicode Zeichen

Wie oben gesehen sind Unicode Zeichen, wie z.B. `∘` als Teil der Julia Syntax erlaubt.

Diese Zeichen können in Julia Notebooks erzeugt werden, indem man den entsprechenden LaTex code eingibt und anschließend "tab" drückt.

Einige Beispiele:

Bezeichne eine Variable als `α`

In [39]:
```
α = 27
```

Out[39]:
```
27
```

In [40]:
```
α
```

Out[40]:
```
27
```

Das `∈`-Zeichen kann `in` ersetzen

In [41]:
```
for j ∈ 1:3
    println(j)
end
```

```
1
2
3
```

Den Vergleichsoperator `>=` können wir schreiben als `≥`

In [42]:
```
if α ≥ 9
    println("α≥9")
end
```

```
α≥9
```

Der Operator `≈` ist äquivalent zur `isapprox()` Funktion:

In [43]:
```
A≈B
```

Out[43]:
```
true
```

Warnung:

1. Unicode Zeichen sind in anderen Editoren unter Umständen nicht so leicht einzufügen.
2. Die großzügige Verwendung von Unicode Zeichen ist eine Besonderheit von Julia. In vielen anderen Programmiersprachen ist das nicht möglich.