Computerphysik Programmiertutorial 1

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: "Hello world!", Variablen und elementare Operationen, Datentypen, Strings, Arrays, Dictionaries, Hilfe, Fehlermeldungen

Hello world!

Essenziell für den Erfolg beim Erlernen einer neuen Programmiersprache ist es, als erstes ein Programm zu schreiben, das "Hello world!" ausgibt:

In [1]:
```
println("Hello world!")
```
```
Hello world!
```

Variablen und elementare Operationen

Variablen sind "Behälter" für Daten. Variablen haben einen Namen und sind mit einem bestimmten Ort im Speicher des Computers verknüpft, an dem die zugehörigen Daten liegen.

Der erste Schritt ist die Zuweisung eines Wertes:

In [2]:
```
a = 42
```
Out[2]:
```
42
```

Nun können wir uns durch Ausgabe vergewissern, dass `a` den Wert `42` hat:

In [3]:
```
a
```
Out[3]:
```
42
```

In [4]:
```
println("Die Variable a hat den Wert ", a)
```
```
Die Variable a hat den Wert 42
```

Julia ist sehr großzügig bei den erlaubten Variablennamen (im Vergleich zu anderen Programmiersprachen).

Nicht erlaubt sind Namen, die mit Ziffern (`0-9`) oder Operatoren (z.B. `+` oder `*`) beginnen.

Zu vermeiden sind außerdem Namen, die gleichzeitig eine andere Bedeutung haben, z.B. mathematische Funktionen wie `log`, die wir später noch kennen lernen.

Jetzt initialisieren wir noch eine weitere Variable mit dem legalen Namen `klaus`:

In [5]:
```
klaus = 5
```
Out[5]:
```
5
```

Daten, die im Speicher liegen kann der Computer verarbeiten. Wir können ihn zum Beispiel damit rechnen lassen. Grundliegende Rechenoperationen mit zwei Zahlen

- Addition: `+`
- Subtraktion: `-`
- Multiplikation: `*`
- Division: `/`
- Ganzzahlige Division: `÷` oder `div`
- Potenz: `^`
- Modulo: `%`

In [13]:
```
klaus^a
```
Out[13]:
```
1444554559021708921
```

Außerdem sind viele mathematische Funktionen vordefiniert, z.B. `exp`, `log`, `sqrt`, `abs`

In [18]:
```
abs(-klaus)
```
Out[18]:
```
5
```

Eine Zahl, die einer Variablen direkt vorgestellt ist, impliziert Multiplikation:

In [19]:
```
2klaus
```
Out[19]:
```
10
```

Datentypen

Jede Variable ist von einem bestimmten Datentyp, den wir mit `typeof` feststellen können:

In [20]:
```
typeof(a)
```
Out[20]:
```
Int64
```

Neben den Ganzzahlentypen (`Int`) gibt es z.B. auch

- Fließkommazahlen `Float`
- komplexe Zahlen `Complex`
- Wahrheitswerte `Bool`
- Textzeichen `Char`

In [21]:
```
kommazahl=2.3
typeof(kommazahl)
```
Out[21]:
```
Float64
```

In [23]:
```
komplexe_zahl=1.3+2im
typeof(komplexe_zahl)
```
Out[23]:
```
ComplexF64 (alias for Complex{Float64})
```

In [25]:
```
wahr=false
typeof(wahr)
```
Out[25]:
```
Bool
```

In [26]:
```
c='c'
typeof(c)
```
Out[26]:
```
Char
```

In Julia ist es beim Programmieren von einfachen Anwendungen oft nicht wichtig sich der Variablentypen bewusst zu sein. Für den Computer ist es aber sehr wichtig, dass mit der Definition jeder Variable ein Typ verabredet wird, da unterschiedliche Typen unterschiedlich viel Speicher belegen.

Im Speicher sind alle Variablen in Binärcode gespeichert. Diesen Code können wir uns mit `bitstring` ausgeben lassen. So sehen wir zum Beispiel, dass unsere Variable `klaus` vom Typ `Int64` 64 bits (oder 8 bytes) belegt:

In [27]:
```
bitstring(klaus)
```
Out[27]:
```
"0000000000000000000000000000000000000000000000000000000000000101"
```

Unsere Variable `c` vom Typ `Char` belegt dagegen nur 32 bits (4 bytes):

In [28]:
```
bitstring(c)
```
Out[28]:
```
"01100011000000000000000000000000"
```

Strings

`Strings` sind Zeichenketten, also Aneinanderreihungen von Textzeichen (`Chars`).

Zum Erstellen eines `Strings` wird die Zeichenkette mit Anführungszeichen `"` umschlossen:

In [29]:
```
s="Hello world!"
```
Out[29]:
```
"Hello world!"
```

Durch einen Index in eckigen Klammern `[]` können wir auf ein bestimmtes Zeichen im String zugreifen:

In [30]:
```
s[4]
```
Out[30]:
```
'l': ASCII/Unicode U+006C (category Ll: Letter, lowercase)
```

Zwei Strings können durch den `*`-Operator aneinandergefügt werden:

In [31]:
```
s1 = s * " Was geht?!"
```
Out[31]:
```
"Hello world! Was geht?!"
```

Werte von Variablen können durch `$variablenname` in Strings eingefügt werden:

In [32]:
```
"Der Wert von a ist $a"
```
Out[32]:
```
"Der Wert von a ist 42"
```

Arrays

Mit einem *Array* können wir mehrere (gleichartige) Daten in einer Datenstruktur zusammenfassen.

Einer Variable können wir ein Array mit spezifischen Werten als Liste dieser Werte eingefasst in eckigen Klammern `[]` zuweisen:

In [33]:
```
array = [1,2,3]
```
Out[33]:
```
3-element Vector{Int64}:
 1
 2
 3
```

Auf einzelen Elemente des Arrays wird mit eckigen Klammern Zugegriffen (**Achtung:** Indizierung startet mit 1!):

In [35]:
```
array[1]
```
Out[35]:
```
1
```

So kann auch dem Eintrag im Array ein neuer Wert zugewiesen werden:

In [36]:
```
array[2]=7
array
```
Out[36]:
```
3-element Vector{Int64}:
 1
 7
 3
```

Neue Elemente am Anfang oder am Ende des Arrays werden mit `pushfirst!` oder `push!` hinzugefügt:

In [37]:
```
pushfirst!(array,0)
```
Out[37]:
```
4-element Vector{Int64}:
 0
 1
 7
 3
```

In [38]:
```
push!(array,23)
```
Out[38]:
```
5-element Vector{Int64}:
  0
  1
  7
  3
 23
```

Elemente am Anfang oder Ende eines Arrays können mit `popfirst!` oder `pop!` entfernt werden:

In [39]:
```
popfirst!(array)
```
Out[39]:
```
0
```

In [40]:
```
array
```
Out[40]:
```
4-element Vector{Int64}:
  1
  7
  3
 23
```

In [41]:
```
pop!(array)
array
```
Out[41]:
```
3-element Vector{Int64}:
 1
 7
 3
```

Zweidimensionale Arrays können ebenfalls direkt durch eine Liste von Werten in eckigen Klammern erstellt werden. Werte einer Zeile werden mit Leerzeichen getrennt, Zeilen werden voneinander durch Semikolon `;` getrennt:

In [42]:
```
M=[1 2 3; 4 5 6]
```
Out[42]:
```
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

Auf Elemente des zweidimensionalen Arrays kann per Doppelindex zugegriffen werden. Der erste Index ist der *Zeilenindex*, der zweite Index ist der *Spaltenindex*:

In [43]:
```
M[2,3]
```
Out[43]:
```
6
```

Alternativ kann ein einzelner Index verwendet werden. Die Abbildung von Doppel- auf Einzelindex ist dabei

```
Einzelindex = Zeilenindex + Spaltenlänge x Spaltenindex
```

Daran sehen wir, das in Julia die Daten zweidimensionaler Arrays spaltenweise hintereinander im Speicher abgelegt werden. Diese Konvention wird als *column-major* Sortierung von höherdimensionalen Arrays bezeichnet.

In [46]:
```
M[3]
```
Out[46]:
```
2
```

Oft brauchen wir viel größere und höherdimensionale Arrays, die wir nicht wie zuvor manuell initialisieren können. Um solche Arrays zu erstellen gibt es verschiedene Möglichkeiten.

Ein dreidimensionales Array von Nullen mit Dimension 3x27x9:

In [47]:
```
nullen = zeros(3,27,9)
```
Out[47]:
```
3×27×9 Array{Float64, 3}:
[:, :, 1] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 2] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 3] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 4] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 5] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 6] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 7] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 8] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0

[:, :, 9] =
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

Ein Array von Zufallszahlen mit Dimension 4x3

In [48]:
```
zufall=rand(4,3)
```
Out[48]:
```
4×3 Matrix{Float64}:
 0.176998  0.362092  0.914484
 0.997288  0.908791  0.468283
 0.150135  0.390281  0.502578
 0.308807  0.959667  0.943961
```

Ein Array der Dimension 2x3x3, das nicht mit spezifischen Werten initialisiert ist. Hier müssen wir dem Computer mitteilen, welchen Datentyp das Array enthalten soll, damit er entsprechend Speicher reservieren kann.

In [50]:
```
leer = Array{Int64}(undef,2,3,3)
```
Out[50]:
```
2×3×3 Array{Int64, 3}:
[:, :, 1] =
 4  5  5
 4  5  5

[:, :, 2] =
 7  7  9
 7  5  5

[:, :, 3] =
 4  11  5
 4  11  5
```

Arrays in Julia können verschiedene Datentypen beinhalten:

In [51]:
```
mix = [klaus, c, wahr]
```
Out[51]:
```
3-element Vector{Any}:
     5
      'c': ASCII/Unicode U+0063 (category Ll: Letter, lowercase)
 false
```

Dictionaries

Mit einem *Dictionary* können wir mehrere Daten in einer Datenstruktur zusammenfassen und dabei nicht-numerische Schlüssel verwenden:

In [52]:
```
meindictionary = Dict("11" => "elf", "zwölf" => 12)
```
Out[52]:
```
Dict{String, Any} with 2 entries:
  "11"    => "elf"
  "zwölf" => 12
```

In [53]:
```
meindictionary["11"]
```
Out[53]:
```
"elf"
```

In [54]:
```
meindictionary["zwölf"]
```
Out[54]:
```
12
```

Hilfe

Die Funktionen von Julia sind in der Dokumentation erklärt. Zur Dokumentation führen zwei Wege

1. Aufrufen der Webseite https://docs.julialang.org/
2. Aufrufen der Dokumentation direkt in Julia mit `?`, z.B. `?println`

In [55]:
```
?println
```
```
search: println printstyled print sprint isprint
```

Out[55]:

```
println([io::IO], xs...)
```

Print (using `print`) `xs` followed by a newline. If `io` is not supplied, prints to `stdout`.

Examples

```
jldoctest
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello, world")

julia> String(take!(io))
"Hello, world\n"
```

Fehlermeldungen

Eher früher als später produziert ihr eure erste Fehlermeldung beim ausführen von Julia Code.

In [57]:
```
b=27
s + s1
```
```
MethodError: no method matching +(::String, ::String)
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:560

Stacktrace:
 [1] top-level scope
   @ In[57]:2
 [2] eval
   @ ./boot.jl:360 [inlined]
 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base ./loading.jl:1094
```

Das ist kein Grund zur Panik! Die Fehlermeldung erklärt **was** schief gelaufen ist und **wo** der Fehler aufgetreten ist.

Falls ein Fehler auftritt, könnt ihr ihn in der Regel beheben, indem ihr folgenden Algorithmus ausführt:

1. Finde die Zeile, in der der Fehler auftritt (**wo**).
2. Versuche anhand Fehlermeldung und des Programmkontexts zu verstehen **was** falsch ist. Falls verstanden, gehe zu Punkt 5.
3. Kopiere die Fehlermeldung in eine Suchmaschine deiner Wahl. Weltweit schreiben Millionen von Menschen Programmcode. Du bist wahrscheinlich nicht die erste, die diesen Fehler produziert. Falls zu den ersten Suchergebnissen ein Forenbeitrag mit gleicher oder ähnlicher Fehlerbeschreibung auftaucht, lies ihn und versuche zu verstehen ob das dein Problem sein könnte. Gute Quellen sind in der Regel z.B. Seiten bei stackoverflow oder github. Falls verstanden, gehe zu Punkt 5.
4. Frage deinen Computerphysik Tutor. Falls **nicht** verstanden, gehe zu Punkt 4.
5. Behebe den Fehler.