1 Erste Schritte mit Julia - live

Bevor Sie dieses Notebook erfolgreich lokal ausführen können, müssen Sie Julia und Pluto einrichten.

Variablen

Wir können eine Variable mit `=` (Zuweisung) definieren. Dann können wir ihren Wert in anderen Ausdrücken verwenden:

```
x = 3
```

```
1 x = 3
```

```
y = 6
```

```
1 y = 2x
```

Standardmäßig zeigt Julia die Ausgabe der letzten Operation an. (Sie können die Ausgabe unterdrücken, indem Sie `;` (ein Semikolon) am Ende hinzufügen.)

Wir können herausfinden, welchen Typ eine Variable hat, indem wir `typeof` benutzen:

```
Int64
```

```
1 typeof(y)
```

Funktionen

Für einfache Funktionen kann eine kurze, einzeilige Funktionsdefinition verwendet werden:

```
f (generic function with 1 method)
```

```
1 f(x) = 2 + x
```

Wenn Sie den Namen der Funktion eingeben, erhalten Sie Informationen über diese Funktion. Um sie aufzurufen, müssen wir Klammern verwenden:

```
f (generic function with 1 method)
```

```
1 f
```

```
12
```

```
1 f(10)
```

Für längere Funktionen verwenden wir die folgende Syntax mit dem Schlüsselwort `function` und `end`:

```
g (generic function with 1 method)
```

```
1 function g(x, y)
2     z = x + y
3     return z^2
4 end
```

```
9
```

```
1 g(1, 2)
```

For-Schleifen

Verwenden Sie `for`, um eine Schleife über eine vorher festgelegte Menge von Werten laufen zu lassen:

```
55
```

```
1 let s = 0
2 
3     for i in 1:10
4         s += i    # Equivalent to s = s + i
5     end
6 
7     s
8 end
```

Hier ist `1:10` ein **Bereich** (range), der die Zahlen von 1 bis 10 darstellt:

```
UnitRange{Int64}
```

```
1 typeof(1:10)
```

Oben haben wir einen `let`-Block verwendet, um eine neue lokale Variable `s` zu definieren. Aber solche Codeblöcke sind normalerweise besser innerhalb von Funktionen aufgehoben, so dass sie wiederverwendet werden können. Beispielsweise könnten wir den obigen Code wie folgt umschreiben:

```
mysum (generic function with 1 method)
```

```
function mysum(n)
    s = 0

    for i in 1:n
        s += i
    end

    return s
end
```

```
5050
```

```
mysum(100)
```

Bedingungen: `if`

Wir können auswerten, ob eine Bedingung wahr ist oder nicht, indem wir einfach die Bedingung schreiben:

```
a = 3
```

```
a = 3
```

```
true
```

```
a < 5
```

Wir sehen, dass Bedingungen einen booleschen Wert (`true` oder `false`) haben.

Wir können dann `if` verwenden, um zu steuern, was wir auf der Grundlage dieses Wertes tun:

```
"small"
```

```
if a < 5
    "small"

else
    "big"

end
```

Beachten Sie, dass das `if` auch den zuletzt ausgewerteten Wert zurückgibt, in diesem Fall die Zeichenkette `"small"` oder `"big"`, Da Pluto reaktiv ist, führt eine Änderung der Definition von `a` oben automatisch dazu, dass dieser Wert neu ausgewertet wird!

Felder (Arrays)

1D Felder (`Vector`)

Wir können einen `Vector` (1-dimensional, oder 1D array) durch eckige Klammern erzeugen:

```
v = [1, 2, 3]
```

```
1 v = [1, 2, 3]
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

```
1 typeof(v)
```

Die `1` im Typ zeigt, dass es sich um ein 1D-Array handelt.

Wir greifen auf die Elemente auch mit eckigen Klammern zu:

```
2
```

```
1 v[2]
```

```
10
```

```
1 v[2] = 10
```

Beachten Sie, dass Pluto die Zellen nicht automatisch aktualisiert, wenn Sie Elemente eines Arrays ändern, aber der Wert ändert sich schon.

Eine gute Möglichkeit, einen Vektor nach einem bestimmten Muster zu erzeugen, ist **array comprehension**:

```
v2 = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
1 v2 = [i^2 for i in 1:10]
```

2D Felder (matrices)

Wir können auch kleine Matrizen (2D-Arrays) mit eckigen Klammern erstellen:

```
M = 2×2 Matrix{Int64}:
 1  2
 3  4
```

```
M = [1 2
     3 4]
```

```
Matrix{Int64} (alias for Array{Int64, 2})
```

```
typeof(M)
```

Das `2` im Typ bestätigt, dass es sich um ein 2D-Array handelt.

Das funktioniert allerdings nicht bei größeren Matrizen. Dafür können wir z.B. verwenden

```
5×5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

```
zeros(5, 5)
```

Beachten Sie, dass `zeros` standardmäßig `Float64` ergibt. Wir können auch einen Typ für die Elemente angeben:

```
4×5 Matrix{Int64}:
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
```

```
zeros(Int, 4, 5)
```

Wir können dann die gewünschten Werte eintragen, indem wir die Elemente manipulieren, z. B. mit einer `for`-Schleife.

Eine schöne alternative Syntax zur Erstellung von Matrizen nach einem bestimmten Muster ist ein *array comprehension* mit einer *doppelten* `for`-Schleife:

```
5×6 Matrix{Int64}:
 2  3  4  5   6   7
 3  4  5  6   7   8
 4  5  6  7   8   9
 5  6  7  8   9  10
 6  7  8  9  10  11
```

```
[i + j for i in 1:5, j in 1:6]
```

Acknowledgement

This notebook is translated from

Computational Thinking, *a live online Julia/Pluto textbook.* (computationalthinking.mit.edu, original notebook)

```
using PlutoUI
```

Inhalt


```
TableOfContents(title="Inhalt")
```