Computerphysik Programmiertutorial 3

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: Zahlenreihen und Tupel, `for`-Schleife, Funktionen, Kommentare, Pakete laden und einfaches Plotten

Zahlenreihen und Tupel

Zahlenreihen und Tupel sind weitere Datenstrukturen, die einzelne Daten zusammenfassen. Beide sind **nicht veränderliche Datenstrukturen**, d.h. einmal erstellt, können ihre Elemente nicht verändert werden.

Mit Zahlenreihen können geordnete Mengen von Zahlen in regelmäßigen abständen erstellt werden. Auf einzelne Elemente wird wie bei Arrays über eckige Klammern `[]` zugegriffen:

In [1]:
```
x=1:10
```
Out[1]:
```
1:10
```

In [2]:
```
x[9]
```
Out[2]:
```
9
```

Zahlenreihen können durch `collect()` zu Arrays konvertiert werden:

In [3]:
```
array=collect(x)
```
Out[3]:
```
10-element Vector{Int64}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
```

In der allgemeinen Syntax

```
<Startwert>:<Schrittweite>:<Maximalwert>
```

kann auch eine Schrittweite angegeben werden:

In [4]:
```
collect(1:2:10)
```
Out[4]:
```
5-element Vector{Int64}:
 1
 3
 5
 7
 9
```

Alternativ können Zahlenreihen mit der Funktion `range()` erstellt werden:

In [5]:
```
collect(range(0,1,step=0.2))
```
Out[5]:
```
6-element Vector{Float64}:
 0.0
 0.2
 0.4
 0.6
 0.8
 1.0
```

Ist das Argument `length` gegeben, produziert `range` eine gleichmäßige Zahlenreihe mit `length` Elementen:

In [6]:
```
collect(range(0,1,length=4))
```
Out[6]:
```
4-element Vector{Float64}:
 0.0
 0.3333333333333333
 0.6666666666666666
 1.0
```

Tupel werden durch runde Klammern `()` erstellt.

In [7]:
```
mein_tupel=("Mo", 23, true)
```
Out[7]:
```
("Mo", 23, true)
```

Auf einzelne Elemente wird durch eckige Klammern `[]` zugegriffen:

In [8]:
```
mein_tupel[2]
```
Out[8]:
```
23
```

`for`-Schleife

In einer `for`-Schleife wird der Anweisungsblock für jedes Element einer iterierbaren Datenstruktur (Array, Zahlenreihe, Tupel, ...) durchgeführt.

Die Syntax ist

```
for <Variable> in <iterierbares Objekt>
    <Anweisungsblock>
end
```

Eine sehr häufige Anwendung ist das Iterieren von Zahlenreihen:

In [9]:
```
for j in 1:10
    println("j hat den Wert $j")
end
```
```
j hat den Wert 1
j hat den Wert 2
j hat den Wert 3
j hat den Wert 4
j hat den Wert 5
j hat den Wert 6
j hat den Wert 7
j hat den Wert 8
j hat den Wert 9
j hat den Wert 10
```

In [10]:
```
for j in 10:-1:0
    println("j hat den Wert $j")
end
```
```
j hat den Wert 10
j hat den Wert 9
j hat den Wert 8
j hat den Wert 7
j hat den Wert 6
j hat den Wert 5
j hat den Wert 4
j hat den Wert 3
j hat den Wert 2
j hat den Wert 1
j hat den Wert 0
```

Genauso können z.B. Tupel iteriert werden:

In [11]:
```
for el in mein_tupel
    println(el)
end
```
```
Mo
23
true
```

Funktionen

Durch Funktionen werden Programmabschnitte zusammengefasst, die eine bestimmte Aufgabe erfüllen. Dadurch wird der Code strukturiert und die wiederholte Eingabe identischer Abschnitte wird vermieden. Außerdem erweitern Funktionen die möglichen Programmiermuster, z.B. durch *Rekursion*.

Syntax der Funktionsdefinition:

```
function <Funktionenname>(<Argumente>)
    <Anweisungsblock>
    return <Rückgabewert>
end
```

In [12]:
```
function polynom(x)
    y = 3x+4x^2
    return y
end

polynom(2)
```
Out[12]:
```
22
```

Wenn keine Rückgabe erfolgen soll, ist der Rückgabewert `nothing`:

In [13]:
```
function say_hi()
    println("Hi!")
    return nothing
end

say_hi()
```
```
Hi!
```

Wichtig: Argumente werden in Julia nach dem Prinzip **pass-by-sharing** übergeben. Das bedeutet, dass Veränderungen, die im Anweisungsblock der Funktion an veränderlichen Datenstrukturen (z.B. Arrays oder Dictionaries) vorgenommen werden auch außerhalb der Funktion sichtbar sind. Das gilt allerdings *nicht* für die primitiven Datentypen (z.B. `Int` oder `Float`) und die unveränderlichen Datenstrukturen.

In [14]:
```
a=(3,4)

function f(x)
    println("x ist ", x)
    x=(1,2,3)
    println("x ist jetzt ", x)
    return nothing
end

f(a)
println("a ist ", a)
```
```
x ist (3, 4)
x ist jetzt (1, 2, 3)
a ist (3, 4)
```

In Julia ist es *Konvention*, dass bei Funktionen, die Änderungen an den gegebenen Argumenten vornehmen, dem Funktionennamen ein `!` angehängt wird:

In [15]:
```
a=[3,4]

function f!(x)
    println("x ist ", x)
    x[1] = 1
    x[2] = 2
    push!(x,3)
    println("x ist jetzt ", x)
    return nothing
end

f!(a)
println("a ist ", a)
```
```
x ist [3, 4]
x ist jetzt [1, 2, 3]
a ist [1, 2, 3]
```

Funktionen können **mehrere Argumente** haben:

In [16]:
```
function produkt(x,y)
    return x*y
end

produkt(2,3)
```
Out[16]:
```
6
```

Funktionen können **elementweise** angewendet werden. Dazu wird einem definierten Funktionennamen der Punkt `.` angehängt:

In [17]:
```
function quadrat(x)
    return x^2
end

array=collect(1:10)
quadrat.(array)
```
Out[17]:
```
10-element Vector{Int64}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

Funktionen können sich selbst aufrufen. Das ermöglicht eine neue Programmierstruktur: die **Rekursion**

So kann z.B. die Fakultät durch Rekursion berechnet werden, da

$$n! = n\big[(n-1)!\big]$$

und

$$1! = 1$$

In [18]:
```
function fact(n)
    if n<=1
        return 1
    else
        return n * fact(n-1)
    end
end

fact(7)
```
Out[18]:
```
5040
```

Kommentare

Ein **Kommentar** ist Text im Quellcode, der nicht als Teil des Programms interpretiert wird. Dadurch können Anmerkungen in den Code eingefügt werden, die ihn verständlicher machen. Es ist sehr wichtig den eigenen Programmcode zu *kommentieren*, also den wesentlichen Ablauf durch das einfügen von Kommentaren zu erklären.

Syntax:

```
# Dies ist ein einzeiliger Kommentar

#=
Mehrzeilige Kommentare werden mit '#=' und '=#' eingeschlossen.
=#
```

So können wir zum Beispiel unsere Fakultätsfunktion kommentieren:

In [19]:
```
#=
Die folgende Funktion berechnet die Fakultät einer Ganzzahl rekursiv.
Argument n: Integer
Rückgabewert: Fakultät von n (n!)
=#
function fact(n)
    if n <= 1
        # Fakultät von 1 ist 1
        return 1
    else
        # Rekursion: n! = n * (n-1)!
        return n * fact(n-1)
    end
end

fact(7)
```
Out[19]:
```
5040
```

Pakete laden und einfaches Plotten

Bisher haben wir gesehen wie die Programmierung in Julia auf einer elementaren Ebene Funktioniert. Im Prinzip können wir damit beliebig komplizierte Programme bauen. Allerdings müssen wir zur Lösung typischer Probleme das Rad nicht immer neu erfinden, da oft gebrauchte Funktionalität in Julia bereits in **Paketen** implementiert ist, die wir in unser Programm einbinden können.

Syntax:

```
using <Paketname>
```

Hier laden wir das Paket `Plots`, das Funktionen zum Plotten enthält:

In [20]:
```
using Plots
```

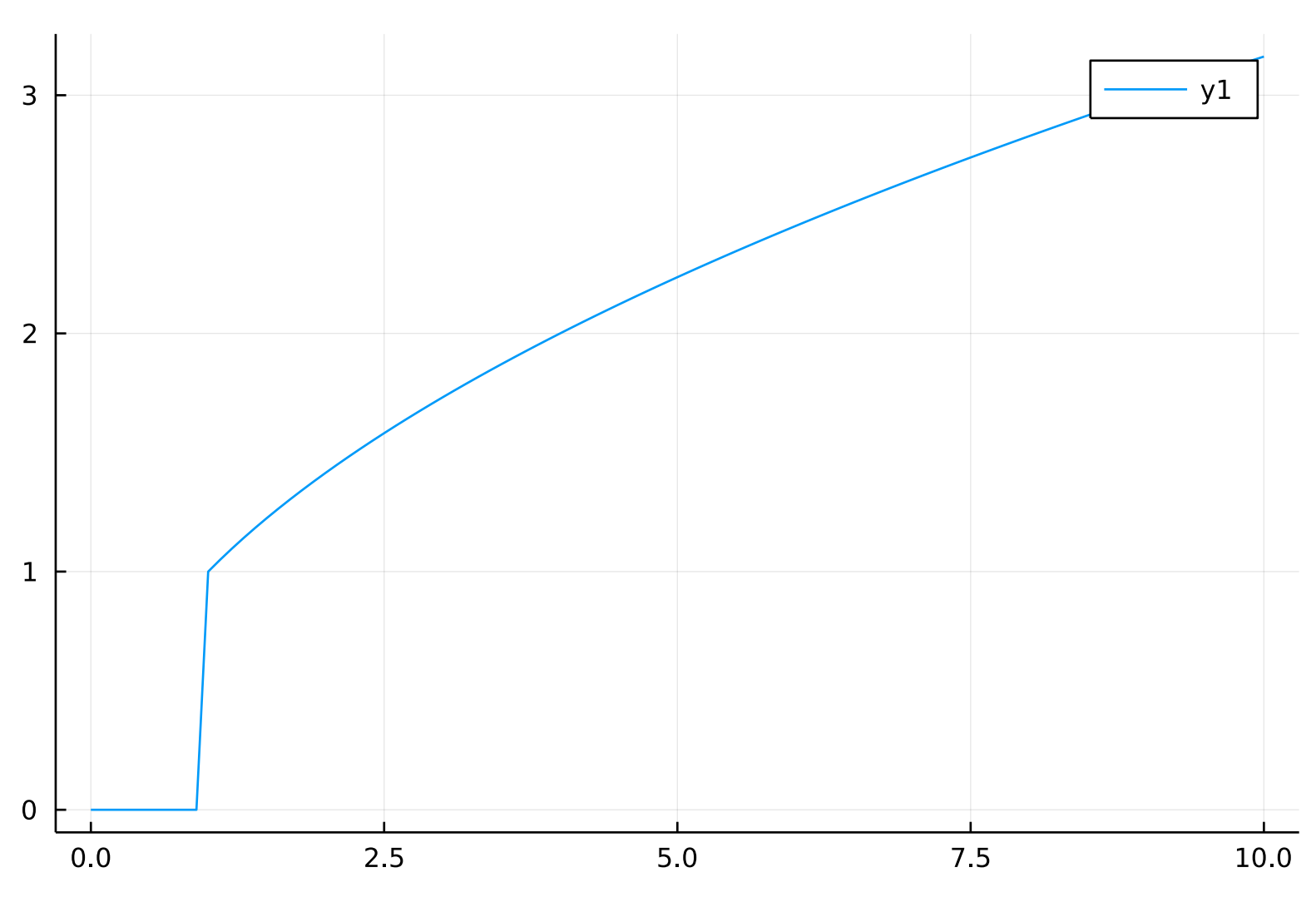
Zum Test definieren wir eine Funktion

In [21]:
```
function f(x)
    if x<1
        return 0.
    end
    return sqrt(x)
end
```
Out[21]:
```
f (generic function with 1 method)
```

Generiere Daten:

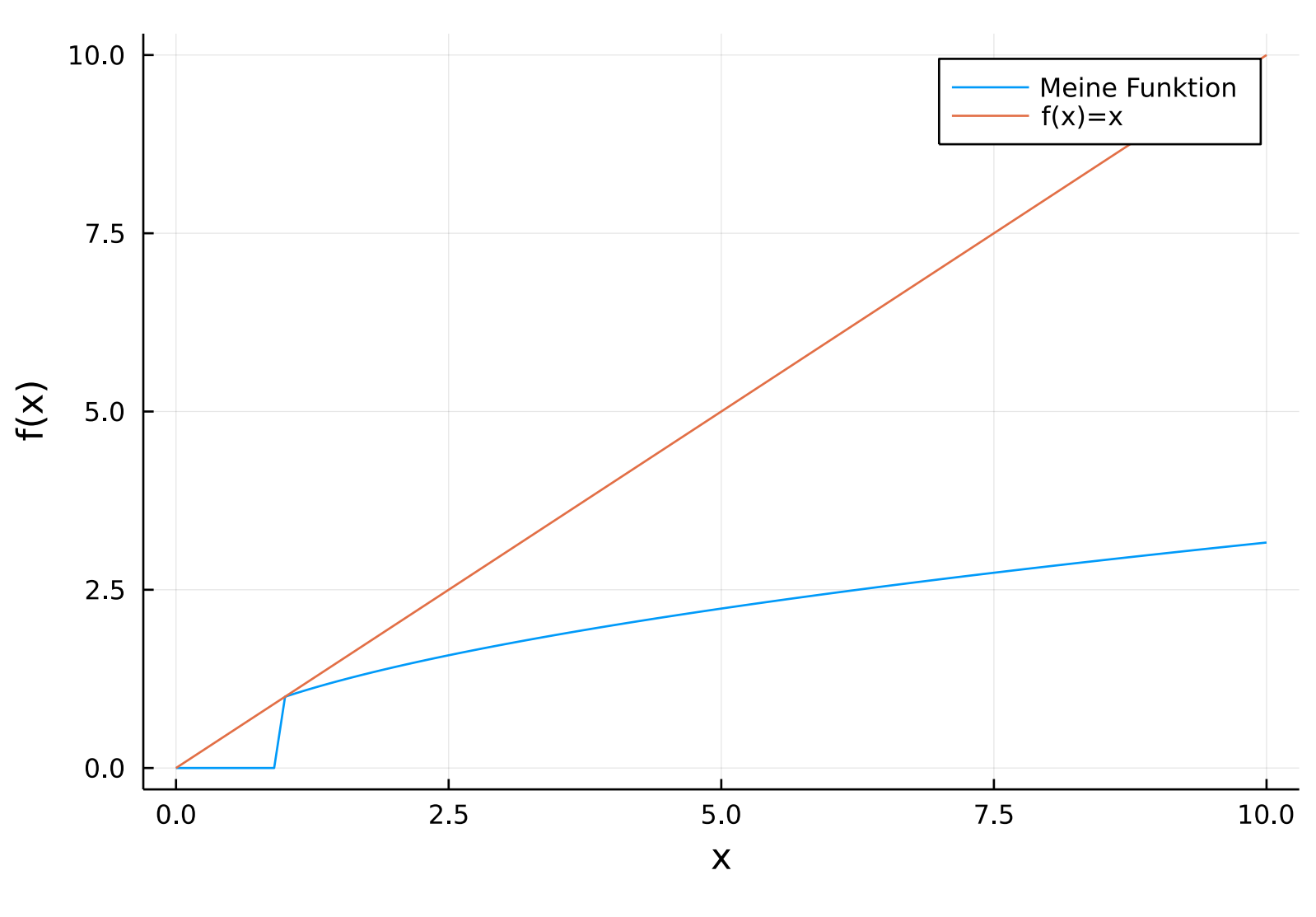
In [22]:
```
x_werte=collect(0:0.1:10)
y_werte=f.(x_werte)
println("x: ", x_werte)
println("y: ", y_werte)
```
```
x: [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0
, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6.0, 6.1,
6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9.0, 9.1, 9.2, 9.
3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9, 10.0]
y: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0488088481701516, 1.0954451150103321, 1.140175425099138, 1.1832159566199232, 1.224744871391589,
1.2649110640673518, 1.3038404810405297, 1.3416407864998738, 1.378404875209022, 1.4142135623730951, 1.449137674618944, 1.4832396974191326, 1.51657508881031, 1
.5491933384829668, 1.5811388300841898, 1.61245154965971, 1.6431676725154984, 1.6733200530681511, 1.70293863659264, 1.7320508075688772, 1.760681686165901, 1.7
888543819998317, 1.816590212458495, 1.8439088914585775, 1.8708286933869707, 1.8973665961010275, 1.9235384061671346, 1.9493588689617927, 1.9748417658131499, 2
.0, 2.0248456731316584, 2.04939015319192, 2.073644135332772, 2.0976176963403033, 2.1213203435596424, 2.1447610589527217, 2.16794833886788, 2.1908902300206643
, 2.2135943621178655, 2.23606797749979, 2.258317958127243, 2.280350850198276, 2.3021728866442674, 2.32379000772445, 2.345207879911715, 2.3664319132398464, 2.
3874672772626644, 2.4083189157584592, 2.4289915602982237, 2.449489742783178, 2.4698178070456938, 2.4899799195977463, 2.5099800796022267, 2.5298221281347035,
2.5495097567963922, 2.569046515733026, 2.588435821108957, 2.6076809620810595, 2.6267851073127395, 2.6457513110645907, 2.6645825188948455, 2.6832815729997477,
2.701851217221259, 2.7202941017470885, 2.7386127875258306, 2.756809750418044, 2.7748873851023217, 2.792848008753788, 2.8106938645110393, 2.8284271247461903,
2.8460498941515415, 2.8635642126552705, 2.8809720581775866, 2.898275349237888, 2.9154759474226504, 2.932575659723036, 2.949576240750525, 2.9664793948382653,
2.9832867780352594, 3.0, 3.0166206257996713, 3.03315017762062, 3.0495901363953815, 3.0659419433511785, 3.082207001484488, 3.0983866769659336, 3.1144823004794
873, 3.1304951684997055, 3.146426544510455, 3.1622776601683795]
```

In [23]:
```
plot(x_werte,y_werte)
```
Out[23]:



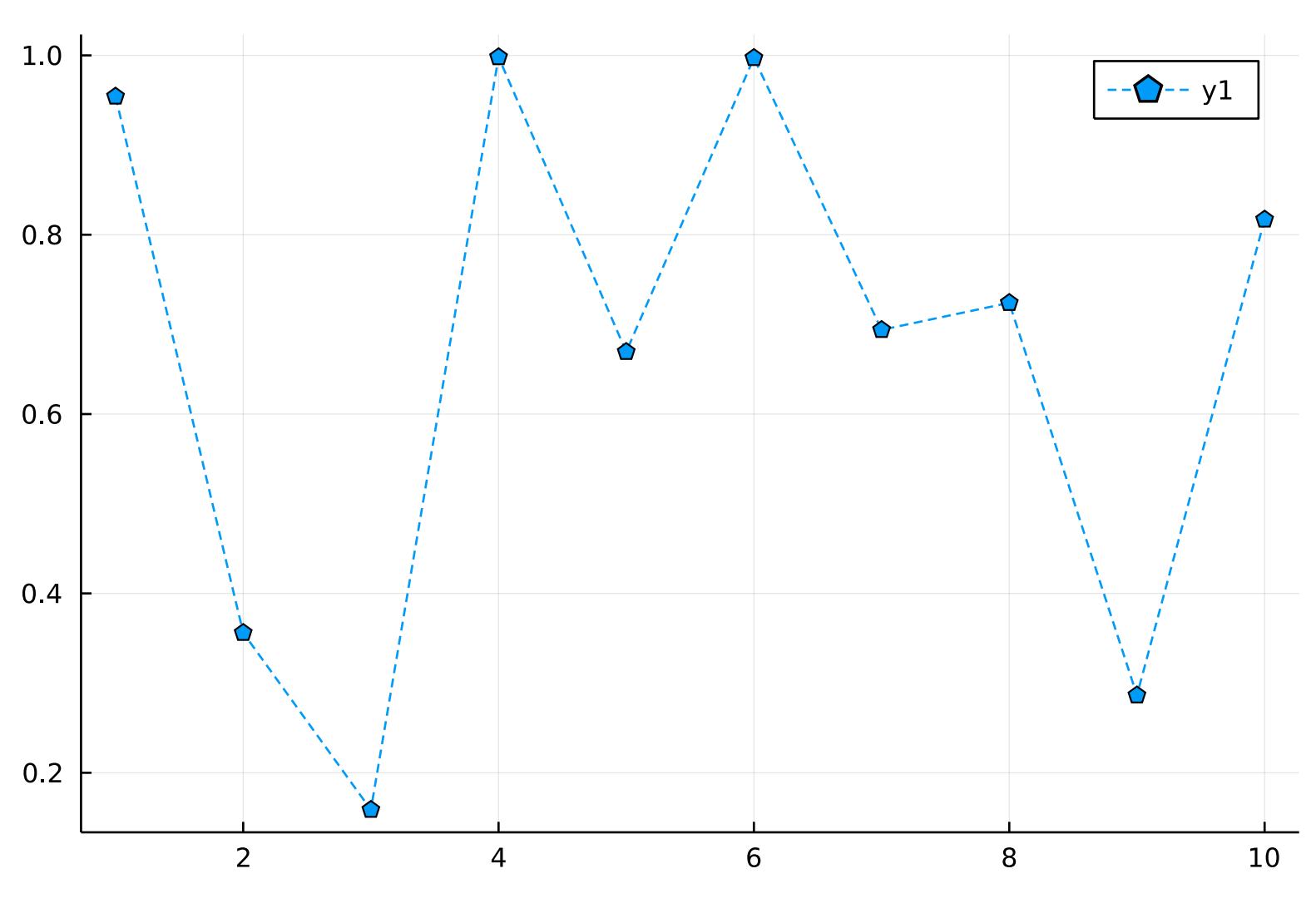
Wir hätten natürlich auch gerne Achsenbeschriftungen:

In [24]:
```
plot(x_werte,y_werte,label="Meine Funktion")
plot!(x_werte,x_werte,label="f(x)=x")
xlabel!("x")
ylabel!("f(x)")
```
Out[24]:



Natürlich können wir auch Datenpunkte markieren oder die Linienart verändern

In [25]:
```
data=rand(10)
plot(data, line=:dash, markershape=:pentagon)
```
Out[25]:



Es gibt zahlreiche Möglichkeiten die Plots zu gestalten: Plots.jl Dokumentation