Python 4 Noobs

(Working Title)

Paul Prechtel, Stefan Brändle

SFZ Ulm

Stand 21. September 2018

- Auslagern und Importieren
- Aufgaben

3. Rekursion
   - Fakultät
   - Fibonacci
   - Türme von Hanoi

Warum Python?

- Es ist einfach!
- *Skriptsprache*
  Python-Code wird nicht kompiliert, sondern von einem anderen Programm interpretiert $\rightarrow$ schneller entwickeln
- *Vielseitig*
- Lässt sich komfortabel mit leistungsstarken, in c geschriebenen Modulen erweitern
  - Imperativ
  - Funktional
  - Aspektorientiert
  - Objektorientiert

Die interaktive Kommandozeilen-Umgebung

```
[user@host ~]$ python3
Python 3.6.5
[GCC 8.0.1 20180317] on linux
Type "help", "copyright", "credits" or "license"
>>>
```

Mit `exit()` oder Strg+D verlässt man die Umgebung.

Das erste Programm

```
>>> print("Hallo, Welt!")
Hallo, Welt!
```

Rechenoperationen

Addition	+
Substraktion	−
Multiplikation	*
Division	/
Modulus (Divisionsrest)	%
Exponential	**

Syntax: Infix-Schreibweise

```
x = a + b # speichert a + b in x
```

Oder auch

```
a += b # addiert b auf a auf
```

Vergleichsoperationen

Ist gleich	==
Ungleich	!=
Größer	>
Kleiner	<
Größer oder gleich	>=
Kleiner oder gleich	<=

Vergleichsoperationen

```
a = 9
b = 7
```

Probiere aus

a == b, a != b, etc...

Boolesche Operationen

a und b	`a and b`
a oder b	`a or b`
Nicht a	`not a`
a gleich b	`a == b`
a ungleich b	`a != b`

Meistbenutzte Datentypen

String	Zeichenfolge
Integer	Ganzzahl
Float	Kommazahl
Boolean	`True` oder `False`
Liste	Beliebige Folge von Werten
None	enthält nichts

Wichtige Funktionen

`print(x)`	gibt x aus
`input()`	gibt Benutzereingabe als String zurück
`type(x)`	gibt den Typ einer Variable zurück
`int(x)`	wandelt x in Ganzzahl
`float(x)`	wandelt x in Kommazahl
`str(x)`	wandelt x in String

Abfragen

Programmblöcke werden ausgeführt, falls eine gewisse Bedingung erfüllt ist.

Beispiel

```
if hour < 11: print("Guten Morgen!")
```

Der zur Abfrage gehörige Code kann auf zwei Arten ausgerichtet sein.

In einer Zeile

```
if condition: doSomething()
```

In den nachfolgenden Zeilen, mit Tabulator eingerückt

```
if condition:
        print("doing something...")
        doSomething()
```

↓ Funktioniert nicht!

```
if condition: print("doing something")
        doSomething()
```

Dasselbe bei Schleifen und Funktionen.

if, elif und else

```
if age < 6:
        # falls age < 6 == True ist:
        print("Go to kindergarten!")
elif age <= 17:
        # falls die vorherige Abfrage erfolglos
        # war und age <= 17 True ist:
        print("Go to school!")
else:
        # falls keine Abfrage bisher erfolgreich
        # war:
        print("Youre adult, do what you want!")
```

Aufgabe

Schreibe ein Programm `timeofday.py`, welches dich nach der Uhrzeit fragt, und je nach Uhrzeit eine passende Antwort gibt!

```
Wie viel Uhr ist es? (in h)
>>> 13
Mittagessen!
```

While-Schleife

```
boring = "n"
while boring == "n":
        # wird wiederholt, solange
        # boring == "n" True ist.
        print("is this loop boring?")
        print("(y/n):")
        boring = input()
```

For-Schleife

for-Schleife in Kombination mit **range**()

```
for i in range(1,4):
        print(i)
```

Ausgabe

```
1
2
3
```

for-Schleife, die Listen durchläuft

```
names = ["Leo", "Tim", "Hans"]
for name in names:
        print("Hallo, " + name + "!")
```

Ausgabe

```
Hallo, Leo!
Hallo, Tim!
Hallo, Hans!
```

Spezielle Statements in Schleifen

continue bricht den aktuellen Durchlauf ab und fährt mit dem nächsten fort.

```
names = ["Leo", "Fritz", "Tim", "Hans"]
for name in names:
        if name == "Fritz": continue
        print(name)
```

Das Beispiel gibt alle Namen außer "Fritz" aus.

break bricht die Ausführung einer Schleife ganz ab.

```
print("Ich bin dein Echo! Echo... Echo...")
while True:
        input = input()
        if input == "stop": break
        print(input)
```

Diese Schleife wird solange alle Eingaben wieder ausgeben, bis der Benutzer `"stop"` eingibt.

Zahlenraten

Das Programm soll ungefähr so ablaufen:

```
Rate die Zahl!
>>> 20
Zu hoch!
>>> 10
Zu niedrig!
>>> 15
Zu hoch!
>>> 12
Erraten!
```

Eine zufällige Ganzzahl von x bis y kannst du mit `random.randint(x, y)` generieren. Am Anfang der Datei musst du das Paket `random` mit **`import`** `random` einbinden

Lösung

```
import random
number = random.randint(0, 100)
print("Rate die Zahl!")
while True:
        guess = int(input)
        if guess > number: print("Zu hoch!")
        if guess < number: print("Zu niedrig!")
        if guess == number: break
print("Erraten!")
```

Listen

`list = [2, "hallo", 42.0]`	eine Liste von Werten
`list[n]`	das n-te Listenelement (start bei 0)
`list[-n]`	das n-te Element von hinten
`list.append(x)`	hängt ein Element an
`list.clear()`	Entfernt alle Elemente
`len(list)`	Länge der Liste

Listen

`list = [2, "hallo", 42.0]`	eine Liste von Werten
`list[n]`	das n-te Listenelement (start bei 0)
`list[-n]`	das n-te Element von hinten
`list.append(x)`	hängt ein Element an
`list.clear()`	Entfernt alle Elemente
`len(list)`	Länge der Liste

In Listen lässt sich jeder beliebige Datentyp einfügen!
Alle Operationen auf Listen unter
`https://docs.python.org/3/tutorial/datastructures.html`

Funktionen

Funktionen teilen das Programm in kleinere Teilprogramme auf (Verschachtelung).

```
def function(x, y):
        # mache etwas mit x und y
        ...
        # falls die Funktion etwas zurueckgibt:
        return result
# das Ende des Einzugs ist das Ende der Funktion
```

Im weiteren Programm lässt sich **function** verwenden:

```
a = function(x, y)
```

Auslagern und Importieren

Funktionen können in andere Dateien ausgalagert werden.

Einbinden einer Datei `foo.py`

```
import foo
```

Aufrufen einer darin definiertn Funktion `bar()`

```
x = foo.bar(y)
```

Bei längeren Dateinamen sinnvoll

```
import foobarnonsense as f
x = f.bar(y)
```

Lege eine Datei `functions.py` an, in die du ab nun neue Funktionen schreibst!

Fakultät

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

Schreibe in der neuen Datei `functions.py` eine Funktion `fac(n)`, welche $n!$ zurück gibt!

Fakultät

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

Schreibe in der neuen Datei `functions.py` eine Funktion `fac(n)`, welche $n!$ zurück gibt!

```
def fac(n):
        product = 1
        for i in range(1, n+1):
                product *= i
        return product
```

Rekursion

„Um Rekursion zu verstehen, muss man Rekursion verstanden haben.“

Rekursion in der Programmierung

Funktionen rufen sich selbst auf.

Rekursive Fakultät

Schreibe eine rekursive Funktion `rfac(n)`, welche $n!$ rekursiv berechnet!

Ansatz

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

Rekursive Fakultät

Schreibe eine rekursive Funktion `rfac(n)`, welche $n!$ rekursiv berechnet!

Ansatz

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

```
def rfac(n):
        if n = 1: return 1
        return n * rfac(n-1)
```

Fibonacci

$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

n	0	1	2	3	4	5	6	7	⋯
fib(n)	1	1	2	3	5	8	13	21	⋯

Fibonacci

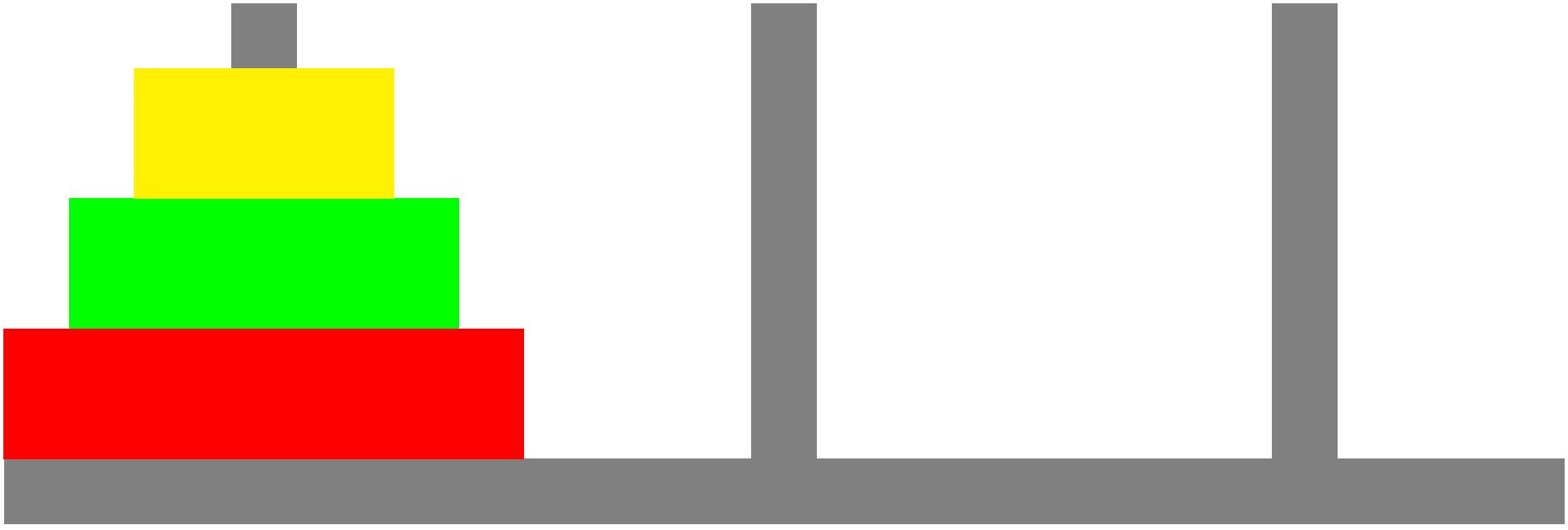
$$fib(0) = 1$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

n	0	1	2	3	4	5	6	7	⋯
fib(n)	1	1	2	3	5	8	13	21	⋯

Schreibe eine Funktion `fib(n)`!

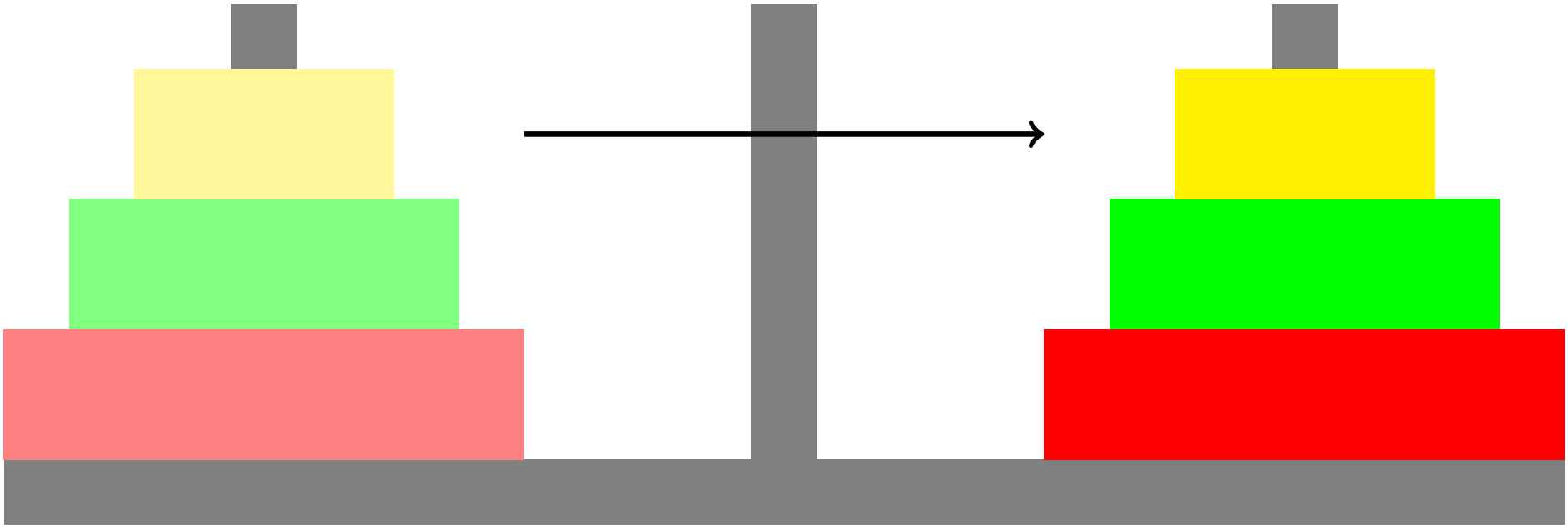
```
def fib(n):
        if (n <= 1): return 1
        return fib(n-1) + fib(n-2)
```

Türme von Hanoi



Die Scheiben müssen einzeln und nacheinander vom ersten auf den letzten Stab verschoben werden.
Auf einer Scheibe darf nur eine kleinere liegen.

Türme von Hanoi



Die Scheiben müssen einzeln und nacheinander vom ersten auf den letzten Stab verschoben werden.
Auf einer Scheibe darf nur eine kleinere liegen.

Lege zuerst eine Datei `hanoi.py` an, welche die Funktion `hanoi(count, src, mid, dest)` enthält.

- `count` ist die Anzahl der zu bewegenden Scheiben
- `src`, `mid`, `dest` ist die Bezeichnung des Ausgangs- Mittel- und Zielturmes. (z.B. `1, 2, 3`)

Diese soll dann auch aufgerufen werden.

Ausgabe

```
[user@host ~]$ python3 hanoi.py
[001] Move 1 to 3
[002] Move 1 to 2
[003] Move 3 to 2
[004] Move 1 to 3
[005] Move 2 to 1
[006] Move 2 to 3
[007] Move 1 to 3
```