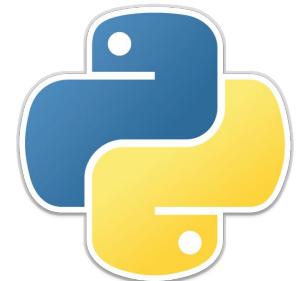
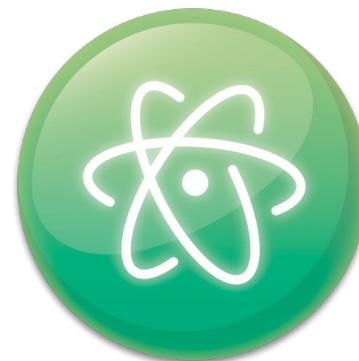
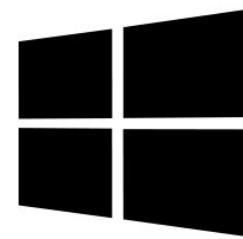
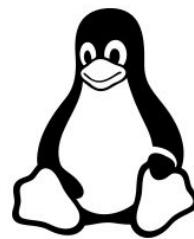
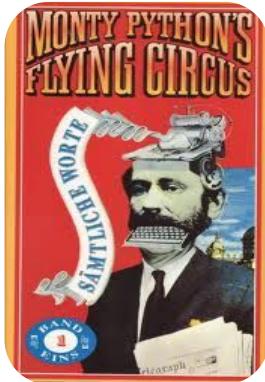




Grundlagen der Programmierung





wichtige Information



Struktur

Dr. Iulian Benta | Tomescu Vlad | Dr. Cătălin Rusu

Workload (in Stunden):

Vorlesung: 2

Seminar/Labor: 2 + 2

MS TEAMS: 7hl9agw (Grundlagen der Programmierung)

Email: rusu@cs.ubbcluj.ro

Fragen und Feedback sind immer erwünscht



Prüfungsform

- **Klausur** (20%)
- **Zwischenprüfung** (20%)
- **Lab** (30%)
- **Praktische Prüfung** (30%)

Minimale Leistungsstandards

K, Z, P, L >= 5 (inkl. aller Labors)



Anwesenheit

- Seminar: **10/14**
 - Labor: **12/14**
 - Kurs: ...
-
- ohne diese Anwesenheit darf man keinen Klausur ablegen
 - man darf das Seminar nur mit seiner Gruppe besuchen
 - Abwesenheiten muss man immer begründen (vom Arzt)



Ziele

- Wie schreibt man Python-Programme
- Grundlegende Konzepte der objektorientierten Programmierung verstehen und selbstständig nutzen
- Herausforderungen und Lösungsansätze des Softwareentwicklungsprozesses verstehen
- Verstehen der Vorgehens- und Denkweisen von Informatikern
- Erster Einblick in Fähigkeiten und Möglichkeiten der Informatik in IT-Projekten



Kursinhalt

- Hello Python
- Software Entwicklung Intro
- Prozedurale Programmierung
- Modulare Programmierung
- Objektorientierte Programmierung
- Softwarearchitektur
- Vereinheitlichte Modellierungssprache (UML)
- Software Testing
- Rekursive Programmierung
- Komplexitätstheorie
- Suchalgorithmen
- Sortieralgorithmen
- Divide-et-Impera
- Backtracking

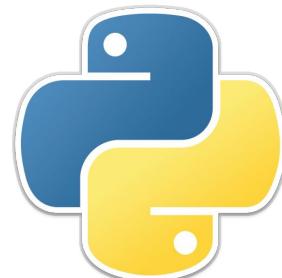


Dieses Semester





1. Hello Python





Was ist Informatik?

Informatik
ist die **Wissenschaft**

von

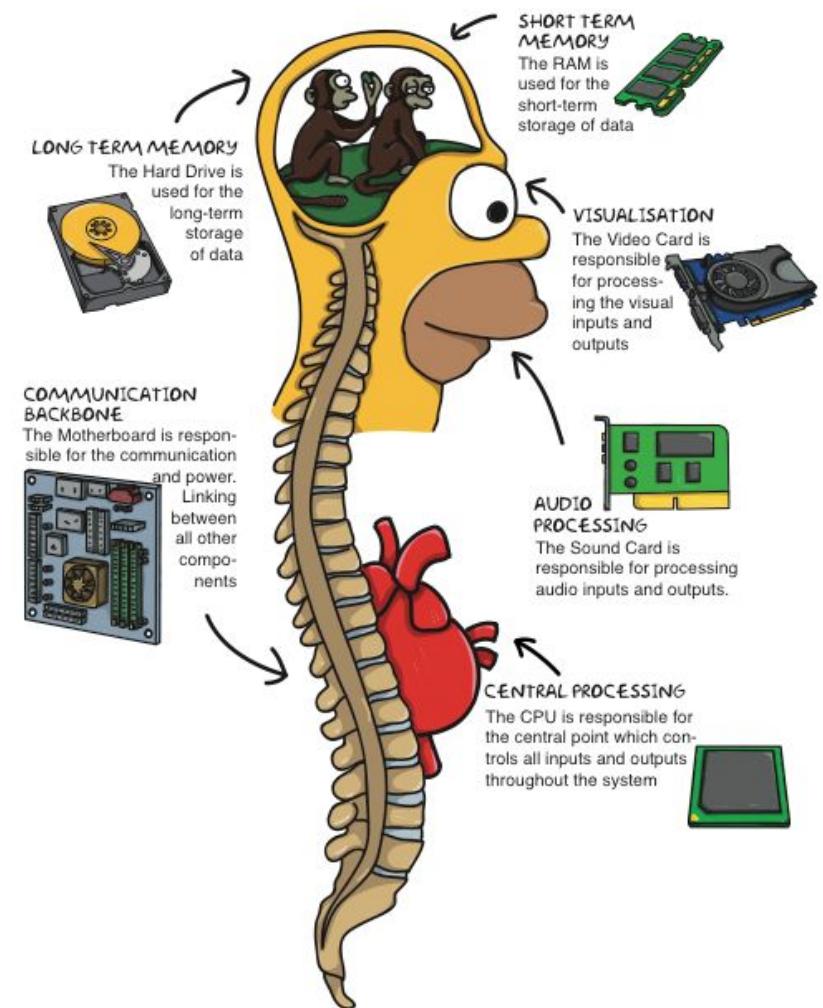


Daten

Daten =
Digitale Repräsentation von Information

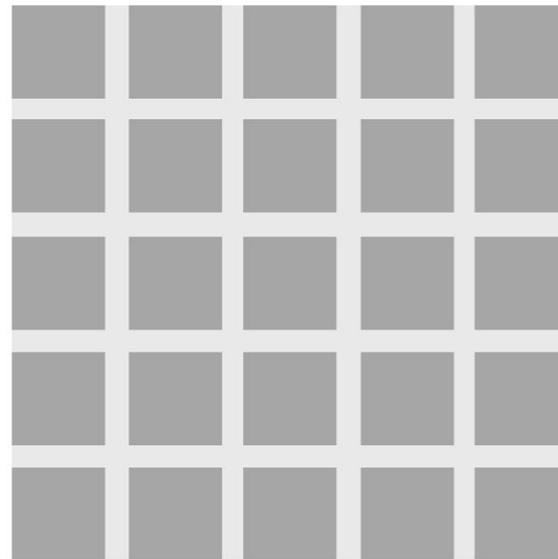
- Strukturierte
 - Semistrukturierte
 - Unstrukturierte
-
- **Datenverarbeitung?**

PROCESSING OF INFORMATION



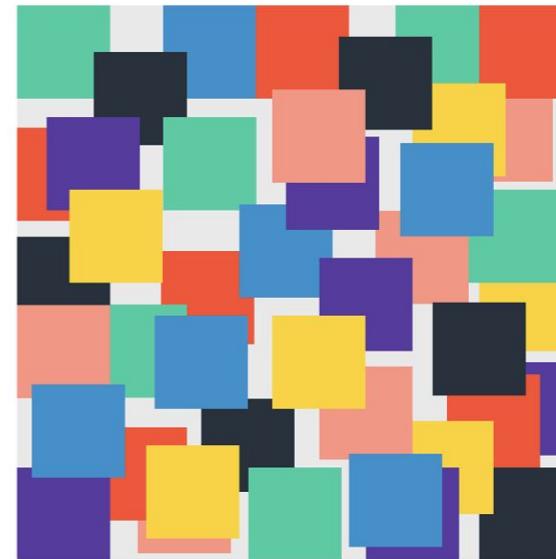
Daten

Structured data



Database, CRM, ERP

Unstructured data



Text, audio, videos



Strukturierte Daten sind so organisiert und formatiert, dass sie in relationalen Datenbanken leicht durchsucht werden können

Unstrukturierte Daten haben kein vordefiniertes Format oder keine vordefinierte Organisation



Was ist Informatik?

Informatik

ist die **Wissenschaft**

von

- der systematischen Verarbeitung und Speicherung von Informationen,
- besonders **der automatischen Verarbeitung mit Hilfe von Computern**

```
        ).html(" "); var b = count_analysed;
        a.parent().find("#"+limit_val).val());
        f = f + f); d < f && (f = d,
        c.length) { for (var g = 0; g < c.length;
        for (g = 0; g < b.splice(e, 1); for (e = 0;
        b.splice(e, 1); if (e.use_class ?
        "#word-list-analysed").html("<span class='count-analysed' title='Top keywords'>" + i + "</span>"); push(d, word), i
        , "
```





Python?

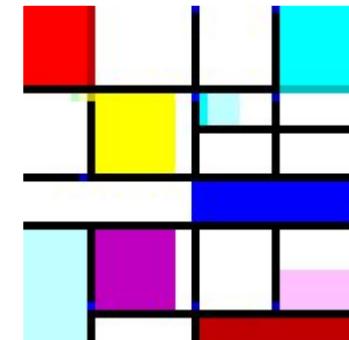
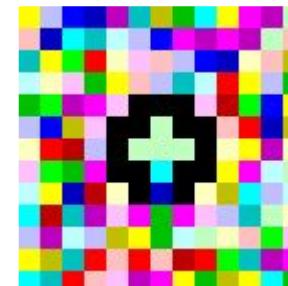
```
whale 🐋 ab 🍇  
pig 🐷 → ab 🍇  
  firstLetter 🖌 0 1  
  rest 🖌 1 🐦 🐦  
apple 🍎 rest firstLetter say 🍎 🍎
```

```
checkbox 🍇  
  😊 pig 🍎 cat 🍎  
  😊 pig 🍎 development 🍎  
  😊 pig 🍎 computer 🍎
```

```
++++++[>+++++>++++++>+++<<<-]>++.>+.++++++  
.++.>++.<<+++++++.>.+++.----.-.-----.>+.
```

IT'S SHOWTIME

TALK TO THE HAND "Hello, World!"
YOU HAVE BEEN TERMINATED





Python?

**SWAP INTEGERS WITHOUT
AN ADDITIONAL VARIABLE**



Overview

Developer Profile

Technology

I. Most Popular Technologies

II. Most Loved, Dreaded, and Wanted

III. Development Environments and Tools

IV. Top Paying Technologies

V. Correlated Technologies

VI. Technology and Society

Work

Community

Methodology

Back to top

Take control of your job search.

Stack Overflow Jobs puts developers first. No recruiter spam or fake job listings.

[Browse jobs](#)

Find your next developer.

Source, attract and recruit developers on the platform they trust most.

[Learn more](#)

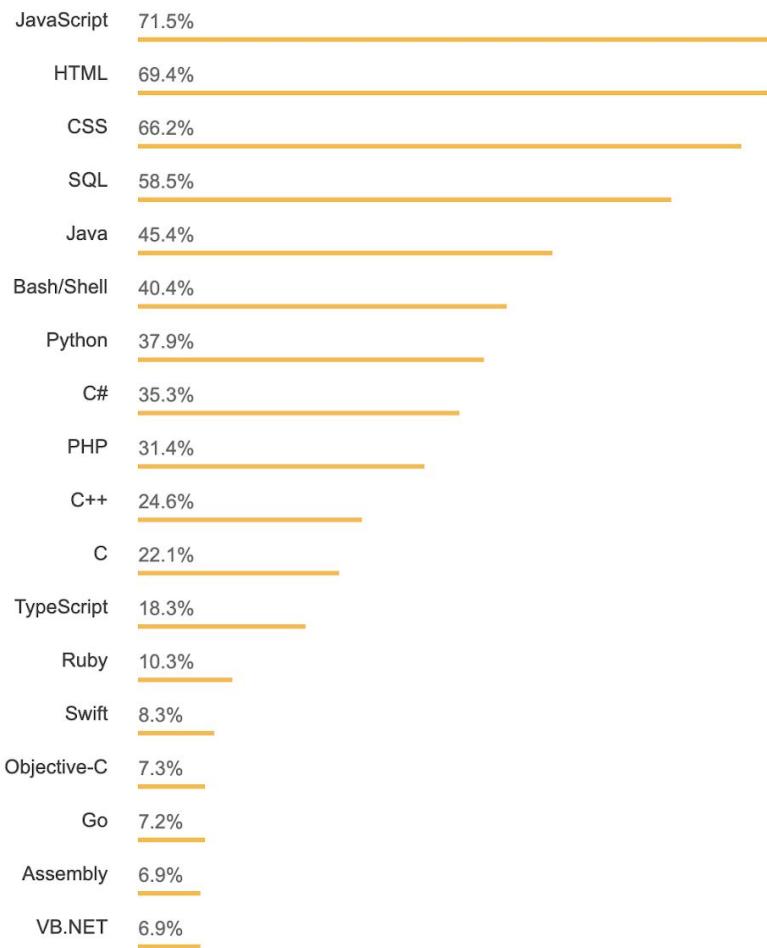


Most Popular Technologies

Programming, Scripting, and Markup Languages

All Respondents

Professional Developers





Python

eine Programmiersprache, welche

- **Einsteigerfreundlich** und **leicht** zu lernen ist
- **Viele Möglichkeiten** bietet ohne unübersichtlich zu werden
- Mehr als ein **Programmierparadigma** unterstützt
- Mit wenigen **Keywords** auskommt



Python ist leicht zu lernen

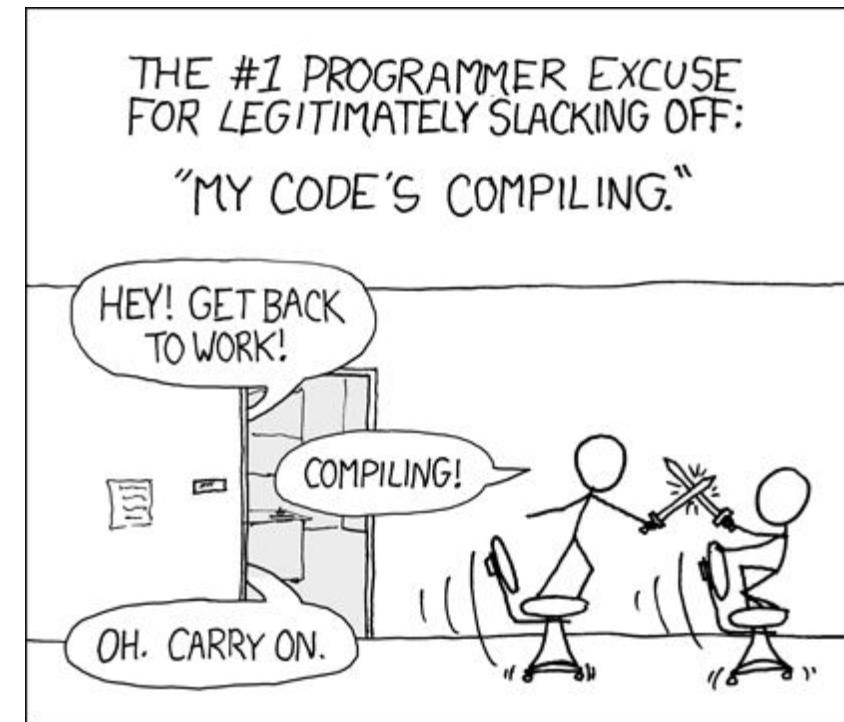
- ist meist wohl strukturiert
- intuitiv
- gut lesbar

```
127     found_pub=crossref_lookup.lookup(xml)
128     if len (found_pub):
129         most_probable = max(found_pub,key=lambda item:item[2])[1]
130
131     return most_probable
132
133 def main ():
134     random.seed(64)
135     prepare_data ()
136
137     pop = toolbox.population(n=50)
138     CXPB, MUTPB, NGEN = 0.5, 0.2, 20
139
140     print("Start of evolution")
141
142     # Evaluate the entire population
143     fitnesses = list(map(toolbox.evaluate, pop))
144     for ind, fit in zip(pop, fitnesses):
145         ind.fitness.values = fit
146
```



Python ist eine Interpretersprache

- mit interaktiver Shell
- erzeugt Python-Bytecode
- nutzt Stackbasierte VM
- gut dokumentiert!



<http://xkcd.com/303/>



Python ist eine moderne Sprache

- Objektorientiert
- Skalierbar
- OS unabhängig
- Reich an Libraries
- Erweiterbar



<http://xkcd.com/138/>



Zen of Python - Ein Mantra

- ***Beautiful*** is better than ugly
- ***Explicit*** is better than implicit
- ***Simple*** is better than complex
- ***Flat*** is better than nested
- ***Sparse*** is better than dense
- ***Readability*** counts
- ...



Python Grundlagen

- man kann Python-Programme interaktiv eintippen (mit der interaktiven Shell)

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

Hallo Welt!

```
>>>
```

```
>>> a = int(input ("a: "))

a: 4

>>> b = int(input ("b: "))

b: 6

>>> c = a + b

>>> print (c)

10

>>>
```



Eigene Syntaxelemente

- **Kommentare:** beginnen mit einem Doppelkreuz-Zeichen #
- **Name:** erlaubt sind die Buchstaben A - Z und a - z, die Zahlen 0 - 9, sowie der Unterstrich "_"
- **Literele:** direkte Darstellung der Werte von Basistypen
>>> STRING = "# Dies ist kein Kommentar."



Grund-Datentypen

- **Integer**
 - `>>> type(1)`
 - `<type 'int'>`
- **(sehr) lange Integer**
 - `>>> type(1L)`
 - `<type 'long'>`
- **Gleitkommazahlen**
 - `>>> type(1.0)`
 - `<type 'float'>`
- **Komplexe Zahlen**
 - `>>> type(1 + 2j)`
 - `<type 'complex'>`
- **Standardoperationen**
 - Addition `+`
 - Subtraction `-`
 - Division `/`
 - Integerdivision `//`
 - Multiplikation `*`
 - Exponentieren `**`
 - Modulo `%`
- **Built-in Funktionen**
 - `round`, `pow`, etc.



Numerische Operationen

Operation	Abkürzung	Vergleichsoperation
• $x = x + y$	$x += y$	<input type="radio"/> $x == y$
• $x = x - y$	$x -= y$	<input type="radio"/> $x != y$
• $x = x * y$	$x *= y$	<input type="radio"/> $x < y$
• $x = x / y$	$x /= y$	<input type="radio"/> $x <= y$
• $x = x \% y$	$x \%= y$	<input type="radio"/> $x > y$
• $x = x^{**}y$	$x **= y$	<input type="radio"/> $x >= y$
• $x = x//y$	$x //= y$	



Wahrheitswerte

- `bool` ist der Typ der Wahrheitswerte `True` und `False`
- Operationen:
 - `not a` (Negation)
 - `a and b` (Konjunktion)
 - `a or b` (Disjunktion)



Ausdrücke und Variablen

Variable: abstrakter Behälter für eine Größe, welche im Verlauf eines Rechenprozesses auftritt

- Name
- Adresse
- in C++: int x;

Python stellt keine Variablen bereit.

Ausdruck: eine Kombination von Operanden (Werten, Variablen) und Operatoren.

```
>>> 2*3-4  
2  
>>> 2*(3-4)  
-2
```



Anweisungen

Programm: eine Abfolge von Anweisungen. Ein Programm ist dabei aus Anweisungsblöcken aufgebaut

Zuweisung: die Verbindung zwischen einem Namen und dem Wert

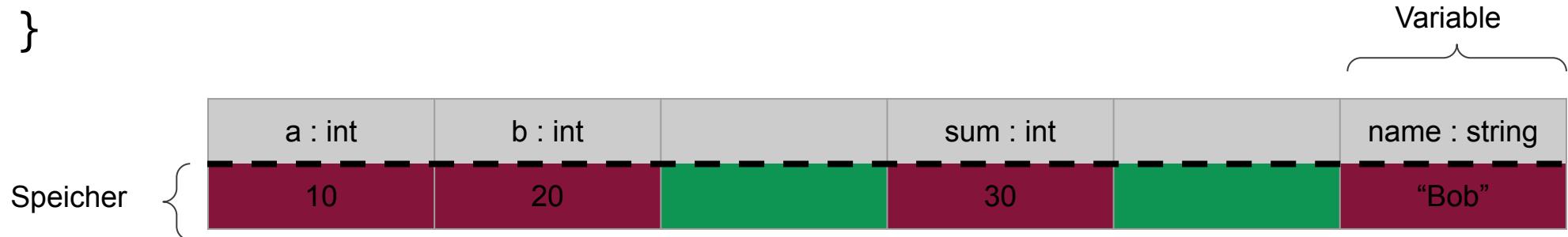
```
>>> x = 1  
>>>  
>>> x = x + 2  
>>>
```

- x ist der Name
- 1 ist ein Objekt vom Typ-Int
- x ist mit einem NEUEN Objekt verbunden, dessen Wert x + 2 ist



Python und Variablen - C++ Beispiel

```
int main () {  
    int a = 10;  
    int b = 20;  
  
    int sum = b + a;  
  
    string name = "Bob";  
  
    std::cout << name;  
  
    a = "Dob" //Fehler (a ist int)  
    float a = 10.0; //Fehler (a existiert schon)  
}
```





Python und Variablen

```
a = 10
```

```
b = 20
```

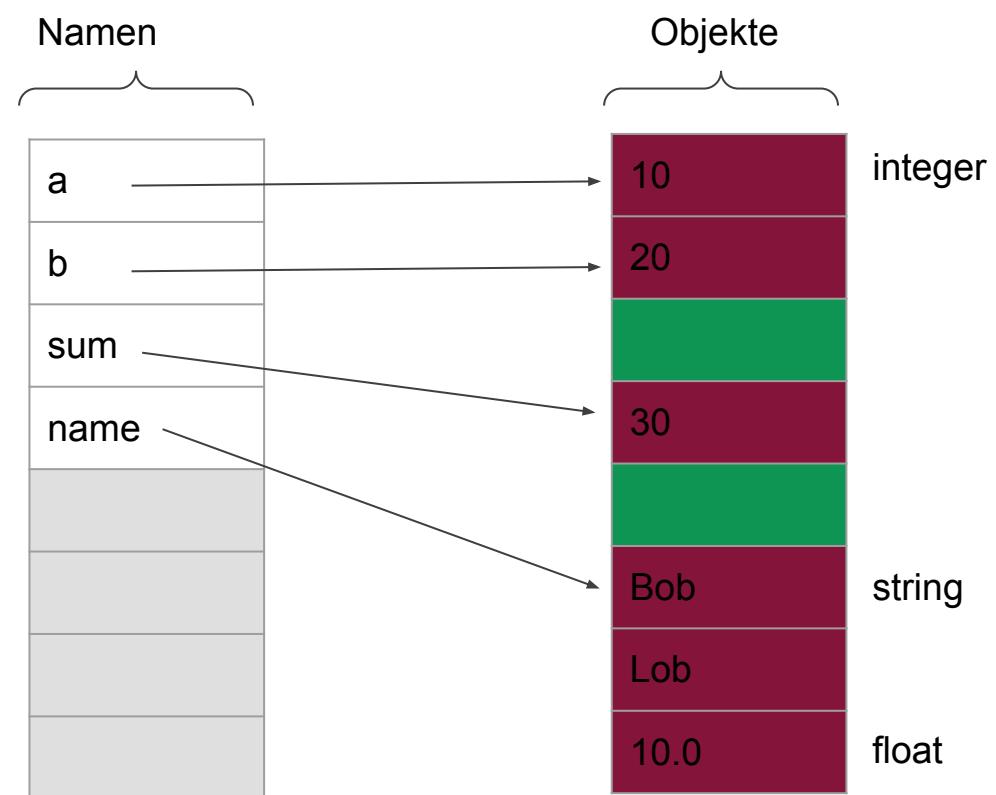
```
sum = b + a
```

```
name = "Bob"
```

```
print (name)
```

```
a = "Dob" #OK
```

```
a = 10.0 #OK
```





Python und Variablen

```
a = 10
```

```
b = 20
```

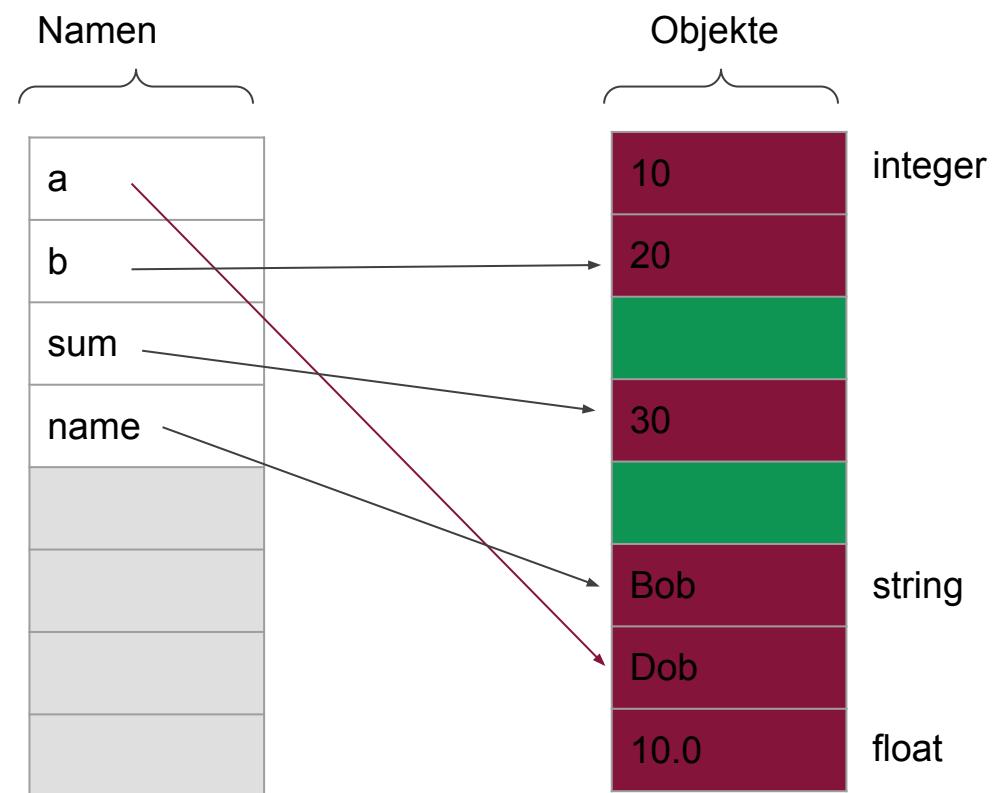
```
sum = b + a
```

```
name = "Bob"
```

```
print (name)
```

```
a = "Dob" #OK
```

```
a = 10.0 #OK
```





Python und Variablen

```
a = 10
```

```
b = 20
```

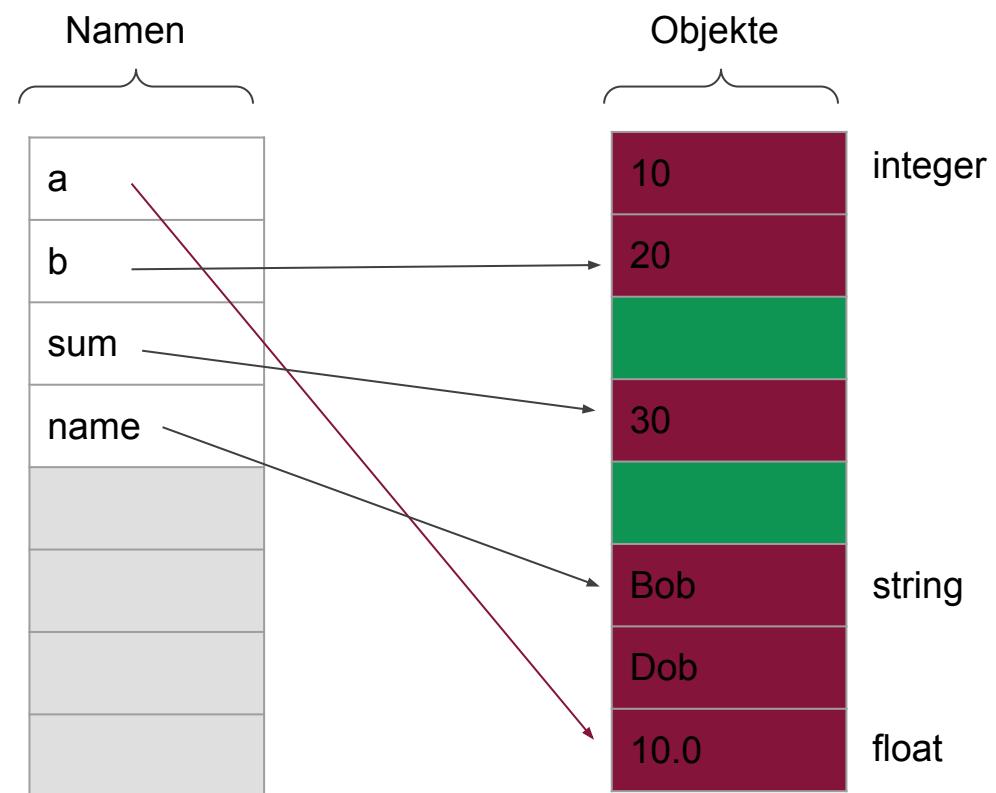
```
sum = b + a
```

```
name = "Bob"
```

```
print (name)
```

```
a = "Dob" #OK
```

```
a = 10.0 #OK
```





If - anweisung

```
if expression1:  
    anweisung1  
[elif expression2: anweisung2]  
    ...  
[else: anweisung]
```

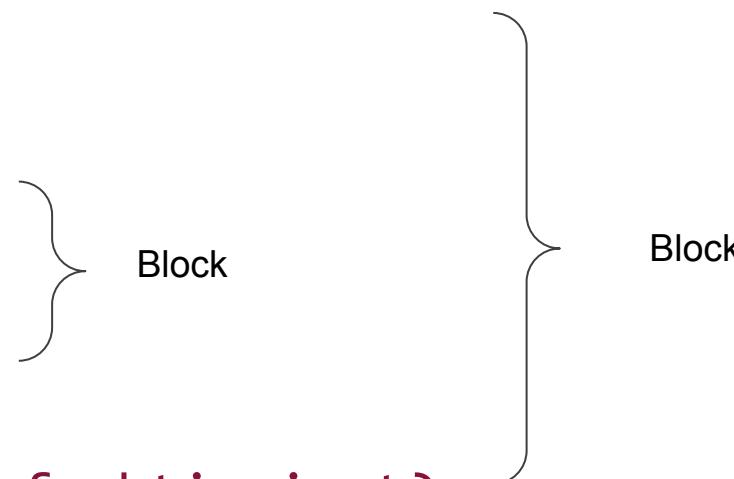
- Die Ausdrücke `expression1`, `expression2`, ... werden in angegebener Reihenfolge ausgewertet
- bis einer zutrifft
 - Dann wird die entsprechende Anweisung ausgeführt.
- Wenn keiner der Ausdrücke zutrifft, wird die `else`-Anweisung ausgeführt.



Einrückungen und Blöcke

- Leerzeichen sind wichtig:
 - die Anweisungen in einer if-Anweisung müssen eingerückt werden
- d.h. die Anweisungen sind zu einem Block gruppiert
- Anweisungen des gleichen Blocks müssen mit der gleichen Anzahl des gleichen Typs Leerzeichen eingerückt sein
- Leerzeichen:
 - Space, Tabulator

```
a = 10
if a == 10:
    b = 20
    print (a+b)
c = 30
print(a)
print(b) #warum funktioniert?
```





While - Schleife

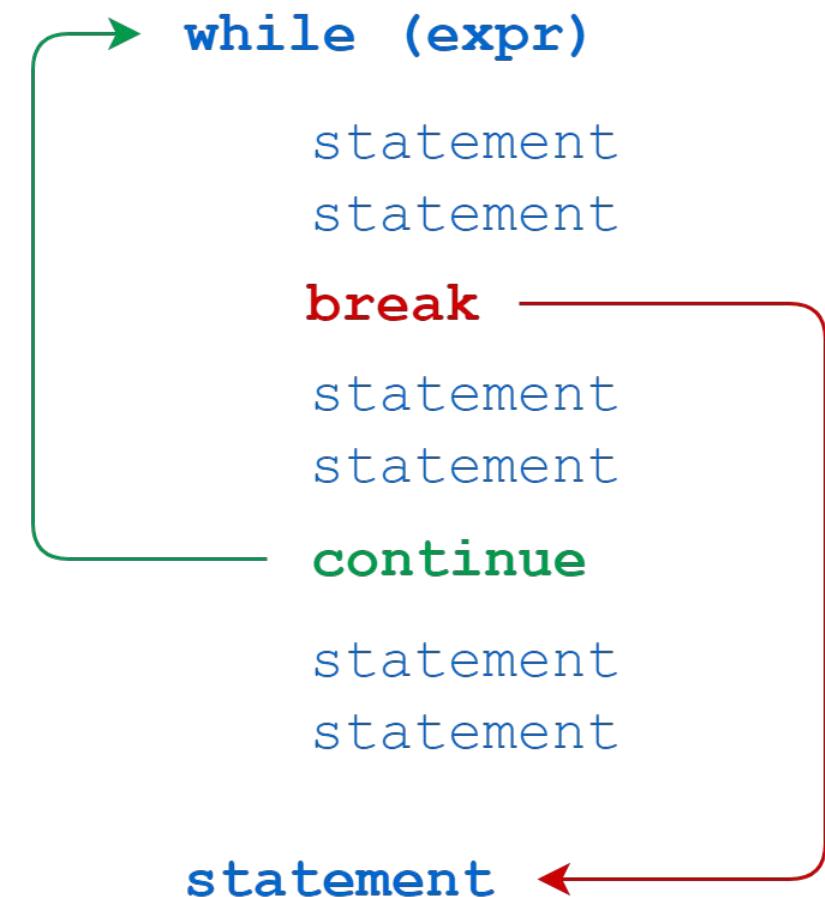
```
while expression:  
    block
```

1. Der Ausdruck `expression` wird ausgewertet
2. Trifft er zu, wird `block` ausgeführt
3. Danach `expression` ist wieder ausgewertet (Schritt 1)



break & continue

- Die break-Anweisung
 - verlässt die aktuelle Schleife
 - expr (die Bedingung) wird nicht ausgewertet
- Die continue-Anweisung
 - überspringt den Rest des Blocks
 - wertet expr neu aus
 - und setzt ggf. die Schleife fort





Sequenzielle Datentypen

Zur Kategorie der sequenziellen Datentypen gehören

- **str** und **unicode** für die Verarbeitung von Zeichenketten
- **list** und **tuple** für die Speicherung beliebiger Instanzen
 - eine **list** nach ihrer Erzeugung verändert werden kann (mutable)
 - ein **tuple** ist nach der Erzeugung nicht mehr veränderbar (immutable)
- **dict** für eine Zuordnung zwischen Objektpaaren
- **set** für ein ungeordneter Zusammenschluss von Elementen, wobei jedes Element nur einmal vorkommen kann



Strings

- `str1 = "abc"`
- `str2 = 'abc'`
- `str3 = """`
 `abc`
 `"""`
- `str4 = ("abc"`
 `"def")`

Escape-Sequenz

- `\a` erzeugt Signalton
- `\b` Backspace
- `\f` Seitenvorschub
- `\n` Linefeed
- `\r` Carriage Return
- `\t` horizontal Tab
- `\v` vertikal Tab
- `\" \\' \\` Escaping " '\\"

```
Catalins-iMac:~ cat$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> name = "bob"
>>> type(name)
<class 'str'>
>>> print(name)
bob
>>> sname = name+"!!"
>>> len(sname)
5
>>> sname
'bob!!'
>>> name[0]
'b'
```



Formatierung mit Strings

Syntax

```
"...%n...%m..." % (Wert1,  
Wert2)
```

Beispiele

```
>>> a = 'H'  
>>> b = 'ello World'  
>>> "%c%s" % (a,b)  
'Hello World'
```

Format

d, i	<i>Integer mit Vorzeichen</i>
f	<i>Float (Dezimaldarstellung)</i>
g, G	<i>Float (wiss. mit Exponent)</i>
u	<i>Integer ohne Vorzeichen</i>
x	<i>Hexzahl ohne Vorzeichen</i>
o	<i>Oktalzahl ohne Vorzeichen</i>
e, E	<i>Float (Exponentendarst.)</i>
c	<i>Zeichen (Länge 1)</i>
s, r	<i>String</i>
%	<i>Prozentzeichen</i>



List

Eine **list** kann Elemente unterschiedlichen Datentyps enthalten

- Syntax `[Wert_1, ..., Wert_n]`
- Eine Liste kann auch nach ihrer Erzeugung verändert werden
- Die Funktion **len()** bestimmt die Anzahl der Elemente der Liste
- Listen können auch Listen enthalten, auch sich selbst
- Hinzugefügt werden Werte mit dem **+ -Operator** und den Funktionen **append()** und **insert()**
- Zugriff auf Elemente mit **[] -Operator**

Tuple



Im Gegensatz zu **Listen** sind **Tuple** immutable

- d.h. jede Änderung erzeugt ein neues Objekt
- Syntax (Wert_1, ..., Wert_n)
- Sie sind damit besonders geeignet, um Konstanten zu repräsentieren
- Ein **Tupel** wird in runde Klammern geschrieben (packing)
- **min()** bestimmt das Minimum eines Tupels, **max()** das Maximum
- Nesting
- unpacking



Tuple vs List

```
[Catalins-iMac:~ cat$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> l = [1,2,3,4]
[>>> l.append(10)
[>>> l
[1, 2, 3, 4, 10]
[>>> l[2] = 33
[>>> l
[1, 2, 33, 4, 10]
[>>> v = l + [101]
[>>> v
[1, 2, 33, 4, 10, 101]
[>>> a = l[2]
[>>> a
33
[>>> t = (1,2,3)
[>>> t.append(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
[>>> t[0] = 101
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
[>>>
```



Dictionary

Mit dict wird eine Zuordnung zwischen Objektpaaren hergestellt

- Syntax { Key_1: Value1, Key_2: Value2, ... }
- müssen die Keys nicht ganze Zahlen (aber Liste?)
- Dictionaries sind iterierbare Objekte
- Die Länge eines Dictionaries `d` kann über `len(d)` abgefragt werden
- Mit `del d[k]` wird das Element mit Schlüssel `k` gelöscht
- mit `k in d` kann geprüft werden, ob sich der Schlüssel `k` in `d` befindet



Dictionary

```
[Catalins-iMac:~ cat$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> d = {}
[>>> d['a'] = 2
[>>> d
{'a': 2}
[>>> d['a']
2
[>>> d.keys()
dict_keys(['a'])
[>>> d.values()
dict_values([2])
[>>> d['b'] = [1,2,3]
[>>> d
{'a': 2, 'b': [1, 2, 3]}
[>>> d['b'][1]
2
[>>> _
```



Set

Eine Menge ist ein ungeordneter Zusammenschluss von Elementen, wobei jedes Element nur einmal vorkommen kann

- Syntax {Wert_1, ..., Wert_n}
- gibt es für mutable Mengen den Typ **set**
- für immutable Mengen den Typ **frozenset**
- `len(m)` liefert die Anzahl der Elemente in m
- `x in m` ist True, wenn x in m enthalten ist
- `m<=t` ist True, wenn m eine Teilmenge von t ist



Set

```
Catalins-iMac:~ cat$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> s = {1,2,3}
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>> l = [1,2,3,4,3
... ]
>>> l
[1, 2, 3, 4, 3]
>>> ns = set(l)
>>> ns
{1, 2, 3, 4}
>>> 2 in ns
True
>>> 5 in ns
False
>>>
```



2. Prozedurale Programmierung





Prozedurale Programmierung

- Strukturierte **Datentypen**
- Was ist eine **Funktion**
- **Wie schreibt man Funktionen in Python**



Strukturierte Datentypen

- weitere Beispiele
- Listen
- Tupel
- Dictionaries



Listen

Operation

s in x

s not in x

x + y

x[n]

x[n:m]

x[n:m:k]

Erklärung

prüft, ob s in x ist

prüft, ob s nicht in x ist

Verkettung von x und y

liefert das n-te Element von x

liefert eine Teilsequenz von n bis m

liefert eine Teilsequenz von n bis m, aber nur jedes k-te Element wird berücksichtigt

len(x)

liefert die Anzahl von Elementen

min(x)

liefert das kleinste Element

max(n)

liefert das größte Element



Listen

```
1 myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 print(myList[:2])
3 print(myList[2:])
4 myList[5:] = ['a', 'b', 'c']
5 print(myList)
6
7 myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
8 myList[1:9] = 'x'
9 print(myList)
10
11
```



Tupel

```
1  tup = 1, 2, 'a'  
2  print(tup)  
3  print(tup[1])  
4  
5  for e in tup:  
6      | | print(e)  
7  
8  ...  
9  | | Was ist die Ausgabe, wenn man diese Zeile auskommentiert?  
10 | ...  
11 #tup[1] = 'x'  
12  
13
```



Dictionaries

```
1  d = {'num':1,'den':2}
2  print(d)
3  print(d['num'])
4  d['num'] = 99
5  print(d['num'])

6
7  if 'num' in d:
8      print('We have num!')
9
10 del d['num']

11
12 if 'num' in d:
13     print('We have num!')
14
15
16
```



Zustand, Verhalten, Identität

Python: alle sind Objekte

Ein Objekt:

- Zustand (state)
 - Verhalten (behavior)
 - Identität
-
- unveränderlichen Grund-Datentypen (Zahlen, Strings, Tupel)...
 - und veränderlichen Objekte Listen, Dictionaries...
-
- `id(Objekt)`
 - `type(Objekt)`
 - `isinstance(Objekt, Typ)`



Zustand, Verhalten, Identität

- in der realen Welt
- wir verwenden täglich viele verschiedene Objekte

Ein Objekt: Laptop

- Zustand (state): Dell (Hersteller), 14" (Bildschirm), Intel (CPU)
 - Eigenschaften
- Verhalten (behavior): anschalten, reset
 - Methoden
- Identität: 8FG89W2 (Serial Number)





Zustand, Verhalten, Identität - in Python

```
1  l = [1,2,3]                                Identität
2
3  print (id(l)) # zB: 4566092872
4
5  v = [1,2,3,4]
6  print (id(l)) # zB: 4566829256          Zustand
7
8  for el in l:
9      print(el) # 1,2,3
10
11 l.append(33)                                verhalten
12 l.pop()
```



unveränderlichen und veränderlichen Objekte

```
1  s = "abc"
2  print(id(s)) #4566030184
3
4  s = s + "d"
5  print(id(s)) #4566832720
6
7
8  l = [1,2,3]
9  print(id(l)) #4566832720
10
11
12  l.append(4)
13  print(id(l)) #4566832720
14
```



unveränderlichen und veränderlichen Objekte

```
1 myList = [1, 2, 3]
2 print(myList)
3 print(myList[1])
4
5 print('Die Liste enthält', len(myList), 'Elemente')
6 print('Das erste Element ist ', myList[0], 'und das letzte ist ', myList[len(myList) - 1])
7
8 x = myList
9 print(myList , x)
10
11 ...
12 | | Das output?
13 ...
14 x[1] = '?'
15 print(myList , x)
16
17
```



unveränderlichen und veränderlichen Objekte

```
1 myList = [1, 2, 3]
2 print(myList)
3 print(myList[1])
4
5 print('Die Liste enthält', len(myList), 'Elemente')
6 print('Das erste Element ist ', myList[0], 'und das letzte ist ', myList[len(myList) - 1])
7
8 x = myList
9 print(myList , x)
10
11 ...
12 | | Das output?
13 ...
14 x[1] = '?'
15 print(myList , x)
16
17
```

- `[1, '?', 3]` `[1, '?', 3]`
- die beiden Listen wurden geändert
- myList und x sind unterschiedliche Name für das gleiche Objekt



unveränderlichen und veränderlichen Objekte

- Unveränderliche Objekte können nach der Erstellung nicht mehr geändert werden
 - d.h. jede Änderung erzeugt ein neues Objekt
- Zugriff auf unveränderliche ist im Prinzip schneller
- Veränderlichen Objekte sind nützlich, wenn die Größe des Objekts geändert werden muss
- Unveränderliche Objekte werden verwendet, wenn man sicherstellen muss, dass das Objekt immer unverändert bleibt
- Unveränderliche Objekte sind grundsätzlich teuer zu „ändern“, da dazu eine Kopie erstellt werden muss.
- Das Ändern veränderlicher Objekte ist billig.



Prozedurale Programmierung

Ein **Programmierparadigma** = ein fundamentaler Programmierstil

Imperative Programmierung: das Programm wird als eine Reihe von Anweisungen geschrieben, die den Zustand des Programms ändern.

Zuweisung: `a = 10`

Prozedurale Programmierung: Programme werden aus einer oder mehreren Prozeduren bzw. Funktionen aufgebaut



Prozedurale Programmierung

Gemäß des **prozeduralen Paradigmas**

- wird der Zustand eines Programms mit Variablen beschrieben
- werden die möglichen Systemabläufe algorithmisch formuliert
- bilden Prozeduren/Funktionen das zentrale Strukturierungs- und Abstraktionsmittel



Prozedurale Programmierung. warum?



- jedes Teil hat ein klar definiertes Ziel
- leicht zu erweitern
- man kann alles verstehen
- Lasagna ist einfach besser :)

- man kann nicht verstehen, wo etwas endet oder etwas anderes beginnt
- schwer zu verstehen, zu erweitern
- ziemlich traurig





Funktionen

Funktion: etwas, das einen oder mehrere Werte nimmt und einen oder mehrere Werte zurückgibt

- Hat einen Namen
- Kann eine Liste von (formalen) Parametern haben
- Kann einen Rückgabewert
- Hat eine Spezifikation

Syntax

```
def <name>(P1, ..., Pn):  
    #anweisungen  
    return <ergebnis>
```

- Definition mit dem Keyword **def**
- man muss mit dem **()-Operator** die Funktion **aufrufen**
- **return** gibt den Wert zurück



Funktionen - Beispiel

```
def absolute_value(num):
    """Diese Funktion gibt den absoluten Wert
       der eingegebenen Zahl zurück"""

    if num >= 0:
        return num
    else:
        return -num

def main():
    print(absolute_value(2))
    print(absolute_value(-4))

main()
```



eine Funktion ohne Spezifikation ist nicht vollständig

```
1 def f (k):
2     v = 2
3     while v < k and k%v:
4         v += 1
5     return v>=k
```

- Könnt ihr bestimmen, was der Code ausgibt?
- Hat es länger als ein paar Sekunden gedauert?
- Jede Funktion hat eine Spezifikation, die besteht aus:
 - Eine kurze Beschreibung
 - Typ und Beschreibung aller Parameter
 - Bedingungen für Eingabeparameter
 - Typ und Beschreibung für den Rückgabewert
 - Bedingungen, die nach der Ausführung erfüllt sein müssen
 - Ausnahmen



Funktionen

```
1 def maximum (x,y ):  
2     """  
3     Gibt das Maximum von zwei Werten zurück  
4     input: x,y – die Parameter  
5     output: der größte der Parameter  
6     Error: TypeError die Parameter dürfen nicht verglichen werden  
7     """  
8     if x>y:  
9         return x  
10    return y  
11
```



Funktionen

Jede Funktion **muss enthalten:**

- sinnvolle Namen (für Parameter und Namen)
- Kommentare
- Eine spezifikation

Testen!

- **man muss jede non-UI Funktion testen (kommt später)**



Übung

Gegeben sei eine Liste L mit $n \geq 2$ positiven Zahlen. Alle Zahlen in der Liste seien unterschiedlich. Schreiben Sie eine Python-Funktion, die das zweitgrößte Element der Liste ausgibt.

Beispiel:

L = [1, 3, 23, 7, 5, 4, 11, 20]

second_best(l) -> 11



Optionale Parameter

```
1 def test(param = 'Hallo'):
2     print (param)
3
4
5 def main():
6     test()
7     test('World!')
8
9 main()
10
11 ...
12 output:
13 Hallo
14 World!
15 ...
16
```



Sichtbarkeit und Blöcke. Teil II

- **Block:** ein Programmabschnitt, der als eine Einheit ausgeführt wird
- Blöcke sind durch einen Einrückungslevel definiert bzw. markiert
- eine Funktion ist ein Block
- ein Block wird innerhalb eines Execution Frame (Aufrufrahmen) ausgeführt
- Wenn eine Funktion aufgerufen wird, wird ein neuer Execution Frame erstellt



Aufrufen einer Funktion (Execution Frame)

Ein Execution Frame enthält:

- Einige administrative Informationen (zum Debugging verwendet)
- Informationen über, wo und wie die Ausführung fortgesetzt wird
- Definiert zwei Namespaces, den **lokalen** und den **globalen** Namespace, die sich auf die Ausführung des Codeblocks auswirken
- Ein Namespace ist eine Zuordnung von Namen zu Objekten.
- Ein bestimmter Namespace kann von mehr als einem Execution Frame referenziert werden



Aufrufen einer Funktion (Execution Frame)

```
1 global_name = 10
2
3 def funktion():
4     local_name = 100
5
6     print(global_name)
7     print(local_name)
8
9     print(locals(), globals())
10
11 funktion()
12
13
```

```
10
100
{'local_name': 100} {'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x7f700c0e7970>, '__spec__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': 'main.py', '__cached__': None, 'global_name': 10, 'funktion': <function funktion at 0x7f700c0cc3a0>}
> []
```



global vs local

```
1 global_name = 10
2
3 def funktion():
4     local_name = 100
5
6     global global_name
7
8     global_name = 101
9
10    print (global_name)
11    print (local_name)
12
13
14
15 funktion()
16 print (global_name)
17
18 ...
19 Output
20
21 101
22 100
23 101
24 ...
25
26
```

```
1 global_name = 10
2
3 def funktion():
4     local_name = 100
5
6     global_name = 101
7
8     print (global_name)
9
10
11
12
13 funktion()
14 print (global_name)
15
16 ...
17 Output
18
19 101
20 100
21 10
22 ...
23
24
25
```



Argumentübergabe

- **Formale Parameter**
 - ein Name für einen Eingabeparameter einer Funktion
 - Jeder Aufruf der Funktion muss für jeden obligatorischen Parameter einen entsprechenden Wert übergeben
- **Tatsächliche Parameter**
 - ein Wert, den der Aufrufer der Funktion für einen formalen Parameter bereitstellt
- Die tatsächlichen Parameter werden beim Aufruf in die lokale Symboltabelle der Funktion eingefügt

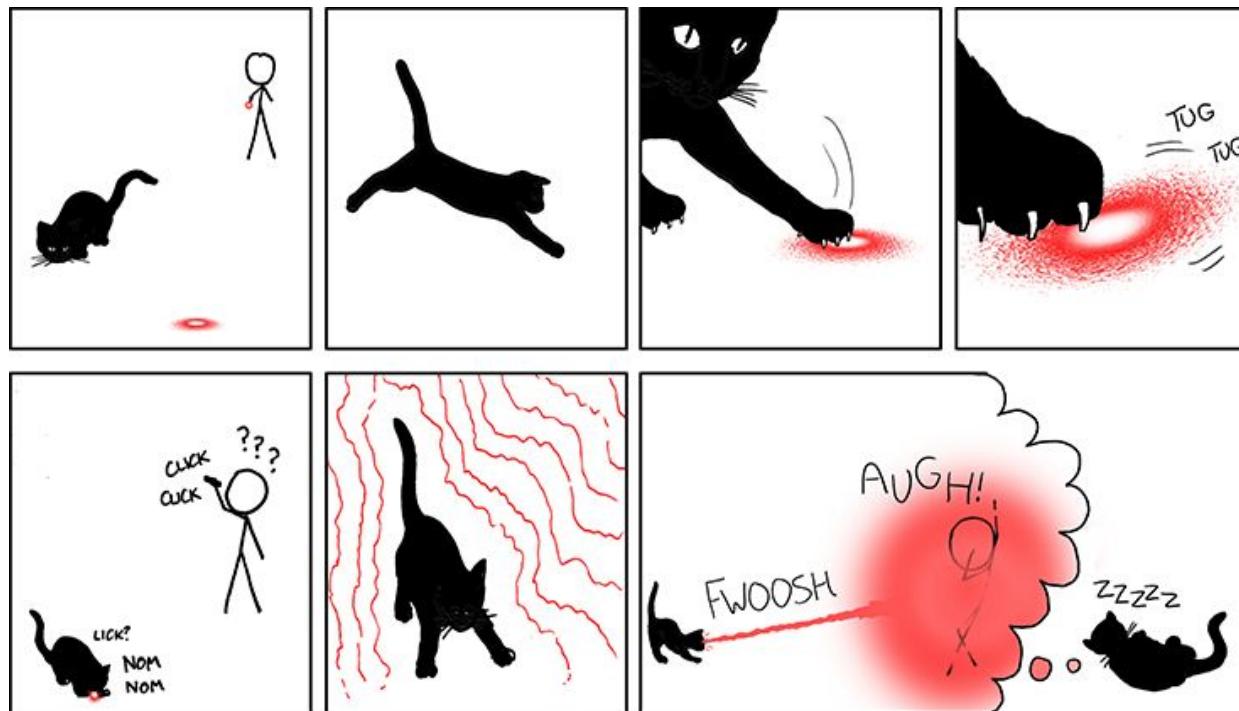
```
1 def test(param = 'Hallo'):  
2     print (param)  
3  
4  
5 def main():  
6     test()  
7     test('World!')
```

Formale Parameter

Tatsächliche Parameter

Argumentübergabe

- die Frage ist: sind Änderungen an einem Parameter für den Aufrufer sichtbar?
- in C++ oder Pascal war die Situation einfach
- man hat **var**, **&**, ***** und andere syntaktische Mechanismen
- aber in Python? leider nicht so einfach :)



<http://xkcd.com/729/>



Argumentübergabe

- **Pass by Value** - eine Kopie des Parameters wird an den formalen Parameter der Funktion gebunden
- **Pass by Reference** - Die Funktion erhält eine Referenz auf das eigentliche Argument statt eine Kopie
- **Side Effect** - Eine Funktion, die die Umgebung des Anrufers ändert (neben der Erzeugung eines Rückgabewerts), soll Side Effects haben



Argumentübergabe

```
1 def refDemo(x):
2     print("2. x=", x, " id=", id(x))
3     x = 42
4     print("3. x=", x, " id=", id(x))
5
6 x = 10
7 print("1. x=", x, " id=", id(x))
8
9 refDemo(x)
10 print("4. x=", x, " id=", id(x))
11
12
```

```
1. x= 10  id= 140197939083264
2. x= 10  id= 140197939083264
3. x= 42  id= 140197939084288
4. x= 10  id= 140197939083264
> |
```

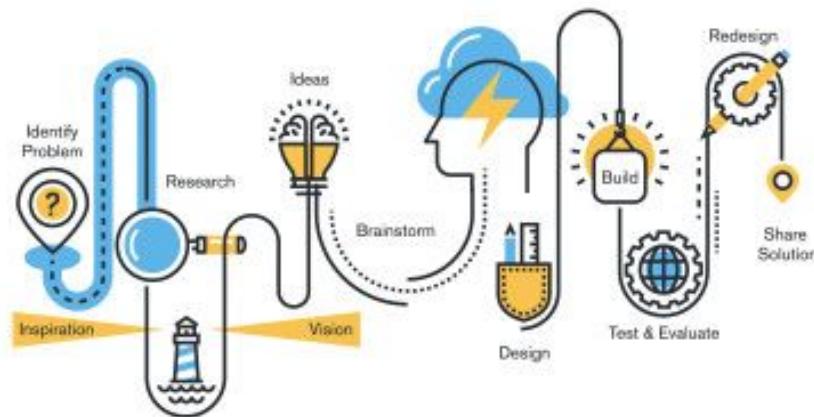


Argumentübergabe

```
1  def noSideEffect(lst):
2      print(lst)
3      lst = [0, 1, 2, 3]
4      print(lst)
5
6  def sideEffect(lst):
7      print(lst)
8      lst += [0, 1, 2, 3]
9      print(lst)
10
11 fib = [0, 1, 1, 2, 3, 5, 8]
12 noSideEffect(fib)
13 # sideEffect(fib)
14 print(fib)
15
16
```

```
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 2, 3]
[0, 1, 1, 2, 3, 5, 8]
> |
```

3. Einführung in Software Engineering





Die Katze auf der Terrasse

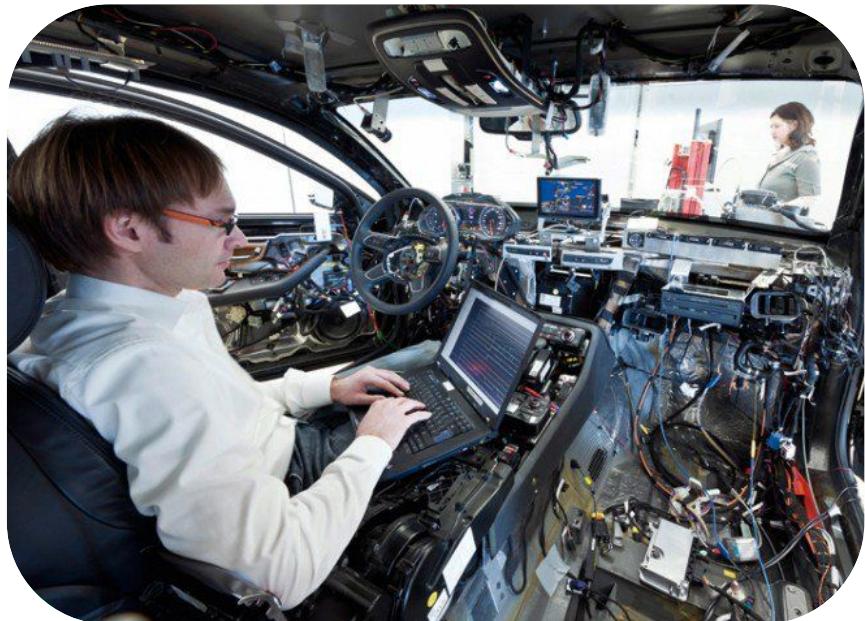
- Mit **Python-Objekten** ist es wie mir der **Katze**,
 - die du irgendwann schlafend auf deiner Terrasse vorfindest.
- Ganz wie ein **Python-Objekten** kann dir die **Katze** nicht sagen, wie sie heißt
 - – **es ist ihr auch ganz egal.**
- Um ihren Namen herauszufinden,
 - wirst du wohl deine Nachbarn fragen müssen,
- und **du** solltest **nicht überrascht** sein,
 - wenn du herausbekommst, dass **die Katze viele Namen hat.**



...es **kann** nützlich sein



Labor 3+





SE != Engineering

Engineering

Software



SE != Engineering

Engineering

- Produkt ist ein physikalisches Objekt

Software

- Produkt ist ein laufendes Programm



SE != Engineering

Engineering

- Produkt ist ein physikalisches Objekt
- Gebaut durch Menschen und Werkzeuge

Software

- Produkt ist ein laufendes Programm
- Gebaut durch Menschen und Werkzeuge



SE != Engineering

Engineering

- Produkt ist ein physikalisches Objekt
- Gebaut durch Menschen und Werkzeuge
- Konstruktion
 - ist teuer

Software

- Produkt ist ein laufendes Programm
- Gebaut durch Menschen und Werkzeuge
- Konstruktion
 - ist billig



SE != Engineering

Engineering

- Produkt ist ein physikalisches Objekt
- Gebaut durch Menschen und Werkzeuge
- Konstruktion
 - ist teuer
 - erfordert Arbeit und Material

Software

- Produkt ist ein laufendes Programm
- Gebaut durch Menschen und Werkzeuge
- Konstruktion
 - ist billig
 - automatisch



SE != Engineering

Engineering

- Produkt ist ein physikalisches Objekt
- Gebaut durch Menschen und Werkzeuge
- Konstruktion
 - ist teuer
 - erfordert Arbeit und Material
 - ist langsam

Software

- Produkt ist ein laufendes Programm
- Gebaut durch Menschen und Werkzeuge
- Konstruktion
 - ist billig
 - automatisch
 - ist schnell



SE != Engineering

Engineering

- Produkt ist ein physikalisches Objekt
- Gebaut durch Menschen und Werkzeuge
- Konstruktion
 - ist teuer
 - erfordert Arbeit und Material
 - ist langsam
 - teuer zu wiederholen

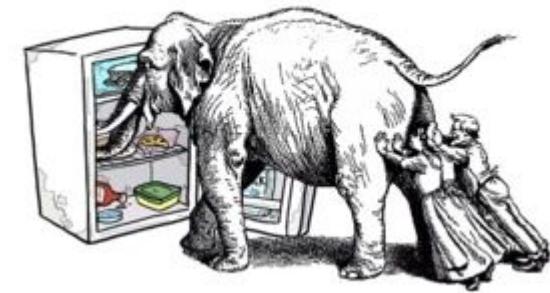
Software

- Produkt ist ein laufendes Programm
- Gebaut durch Menschen und Werkzeuge
- Konstruktion
 - ist billig
 - automatisch
 - ist schnell
 - leicht zu wiederholen





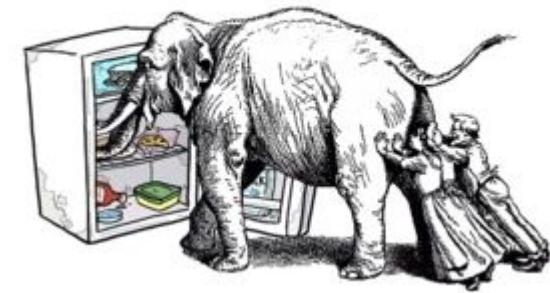
Wie steckt man einen Elefant in einen Kühlschrank?





Wie steckt man einen Elefant in einen Kühlschrank?

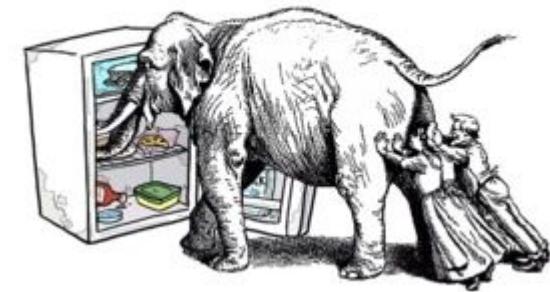
1. Man öffnet den Kühlschrank





Wie steckt man einen Elefant in einen Kühlschrank?

1. Man öffnet den Kühlschrank
2. stellt den Elefanten hinein

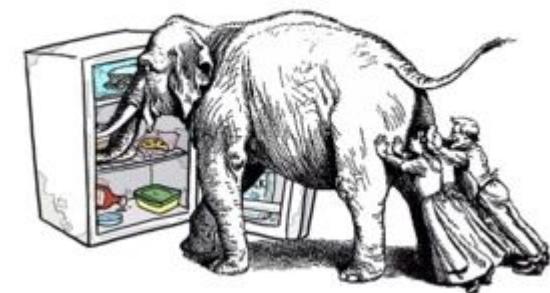




Wie steckt man einen Elefant in einen Kühlschrank?

1. Man öffnet den Kühlschrank
2. stellt den Elefanten hinein
3. schließt die Tür

...aber





wie denkt ein Entwickler?





wie denkt ein Entwickler?

1. teile das Problem in **mehrere Probleme** auf





wie denkt ein Entwickler?

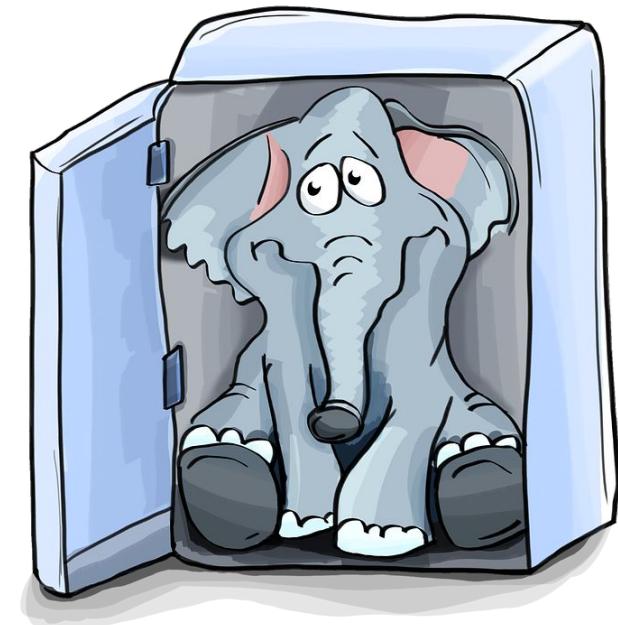
1. teile das Problem in **mehrere Probleme** auf
2. finde **Lösungen** für die kleine Probleme





wie denkt ein Entwickler?

1. teile das Problem in **mehrere Probleme** auf
2. finde **Lösungen** für die kleine Probleme
3. stelle die Lösungen zusammen





wie denkt ein Entwickler?

1. teile das Problem in **mehrere Probleme** auf
2. finde **Lösungen** für die kleine Probleme
3. stelle die Lösungen zusammen
4. Aufräumen (**refactor**)





teile das Problem auf

1. Was für einen Kühlschrank habe ich?
2. Aber der Elefant?
3. Wo finde ich den Elefant?
4. Transport
5. Was könnte ich machen, falls der Elefant zu groß ist?



finde Lösungen

finde Lösungen



1. Was für einen Kühlschrank nutze ich?



finde Lösungen

1. Was für einen Kühlschrank nutze ich? - Mein



finde Lösungen

1. Was für einen Kühlschrank nutze ich? - **Mein**
2. Aber der Elefant?



finde Lösungen

1. Was für einen Kühlschrank nutze ich? - **Mein**
2. Aber der Elefant? - **Afrikanischer**



finde Lösungen

1. Was für einen Kühlschrank nutze ich? - **Mein**
2. Aber der Elefant? - **Afrikanischer**
3. Wo finde ich den Elefant?



finde Lösungen

1. Was für einen Kühlschrank nutze ich? - Mein
2. Aber der Elefant? - Afrikanischer
3. Wo finde ich den Elefant? - Afrika



finde Lösungen

1. Was für einen Kühlschrank nutze ich? - **Mein**
2. Aber der Elefant? - **Afrikanischer**
3. Wo finde ich den Elefant? - **Afrika**
4. Transport



finde Lösungen

1. Was für einen Kühlschrank nutze ich? - **Mein**
2. Aber der Elefant? - **Afrikanischer**
3. Wo finde ich den Elefant? - **Afrika**
4. Transport - mit dem Flugzeug, im Gepäck



finde Lösungen

1. Was für einen Kühlschrank nutze ich? - **Mein**
2. Aber der Elefant? - **Afrikanischer**
3. Wo finde ich den Elefant? - **Afrika**
4. Transport - mit dem Flugzeug, im Gepäck
5. Was könnte ich machen, wenn der Elefant zu groß ist?





stelle die Lösungen zusammen

1. ich leihe ein shrinkgun von Gru
2. fliege nach Südafrika
3. besichtige einen Elephant-Park
4. finde einen Elefant im Park
5. schieße den Elefant mit dem shrinkgun
6. lege den Elefant ins Gepäck
7. fahre zum Flughafen
8. fliege zurück
9. fahre nach Hause
10. stecke den Elefant in den Kühlschrank



und für Programmierung....

Wir brauchen eine Funktion, das die Anzahl von Erscheinungen aller Elemente in einer Liste bestimmt.

Beispiel:

- input: `l = [1, 2, 6, 5, 3, 4, 2, 4, 1]`
- output: `1 - 2, 2 - 2, 3 - 1, 4 - 2, 5 - 1`

Fragen:

- wie kann man das output representieren?

Schritte:

- man muss das Output initialisieren
- man muss alle Elemente der Liste durchgehen
- für ein Element man bestimmt die Anzahl von Erscheinungen
- man fügt die anzahl in Ouput
- man gibt das Ouput zurück



und für Programmierung....

Fragen:

- wie kann man das output representieren?
 - Dictionary

Schritte:

- man muss das Output initialisieren
 - `d = {}`
- man muss alle Elemente der Liste durchgehen
 - `for elem in l:`
- für ein Element man bestimmt die Anzahl von Erscheinungen
 - neue Funktion: `anzahl()`
- man fügt die anzahl in Ouput
 - `d['elem'] = a`
- man gibt das Ouput zurück
 - `return d`



und für Programmierung....

```
l = [1,2,6,5,3,4,2,4,1]
```

```
def my_funk(l):
    d = {}

    for elem in l:
        a = anzahl(elem,l):
        d[‘elem’] = a

    return d
```



und für Programmierung....

jetzt muss man das Gleiche für die Anzahl Funktion machen

```
def anzahl(el, l):  
    a = 0  
  
    for elem in l:  
        if el == elem:  
            a += 1  
  
    return a
```



Vorgehen in SE

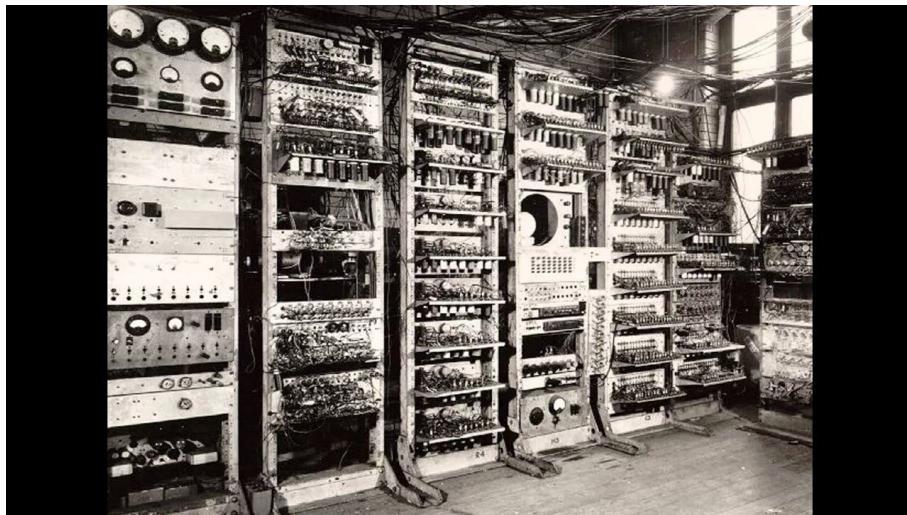
- Wie alle Aktivitäten, die die Zusammenarbeit zwischen Menschen beinhalten
 - Programmierung ist nicht einfach
- Woher kommt die Problembeschreibung?
- Beschreibt sie wirklich das Problem des Nutzers?
- Welche Struktur soll das fertige Programm haben?
- Löst das Programm wirklich das Problem?





a long long time ago....

- einmal war es einfach
- die Welt war irgendwie größer
- Computer waren groß, langsam und Anwendungsfälle waren äußerst begrenzt





Software-Krise

- Mitte 1960er Jahre
- Hardware ist schneller geworden
 - Mit schnellerer Hardware wurde Software wichtiger
 - Als Konsequenz Anforderungen stiegen
- Die einfache Situation von gestern wurde schwer zu managen
- The Humble Programmer
 - <https://cacm.acm.org/magazines/1972/10/11993-the-humble-programmer/pdf>

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

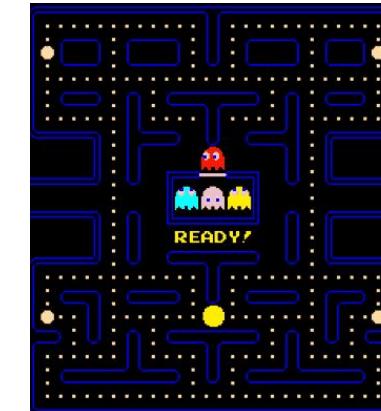
— Edsger Dijkstra, The Humble Programmer (EWD340), Communications of the ACM

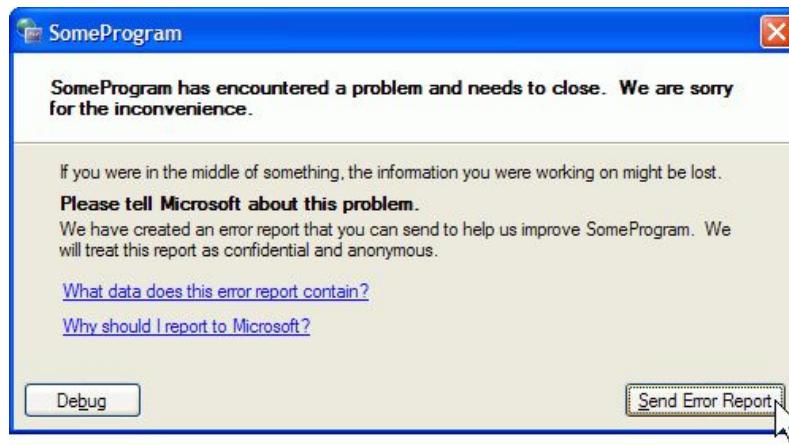
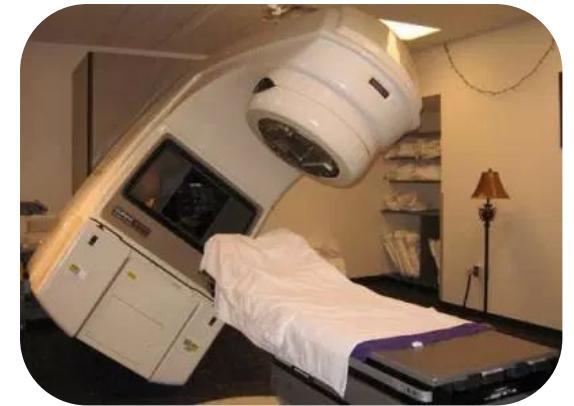
Warum hat Hardware das verursacht?

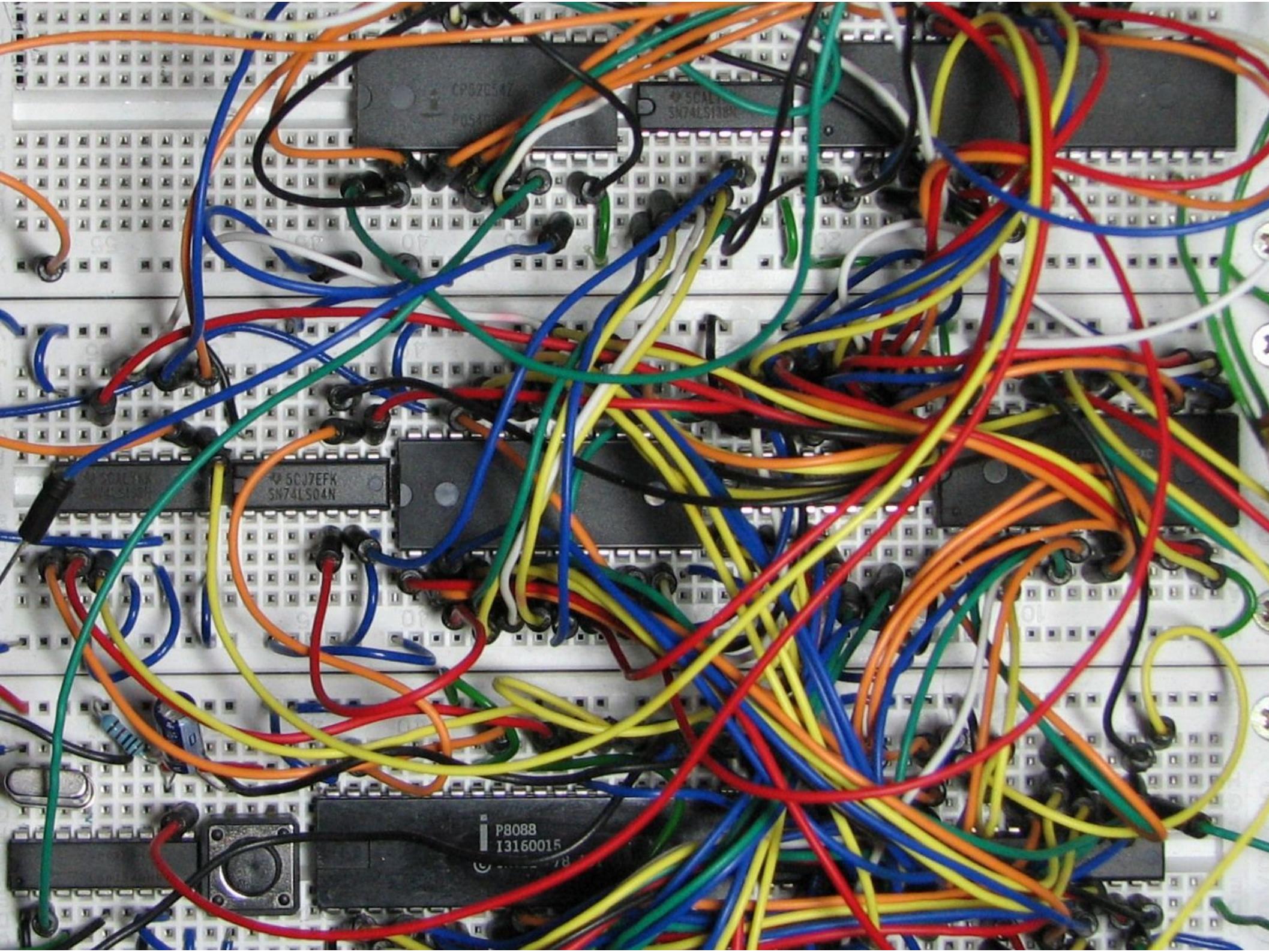
```
def main():
    while true:
        ...

def main():
    game_running=true
    while game_running:
        process_user_input()
        game_world()
        draw_game_world()

    #close game
    if Pressed(KEY_ESCAPE):
        game_running = false
```







- Nicht nur Hardware ist schneller geworden, sondern wir leben in einer vernetzten Welt
- Heute läuft Software auf Handys, Computern, Smartwatches und Haushaltsgeräten
- und beeinflusst alle Aspekte unseres Lebens





Extreme Komplexität

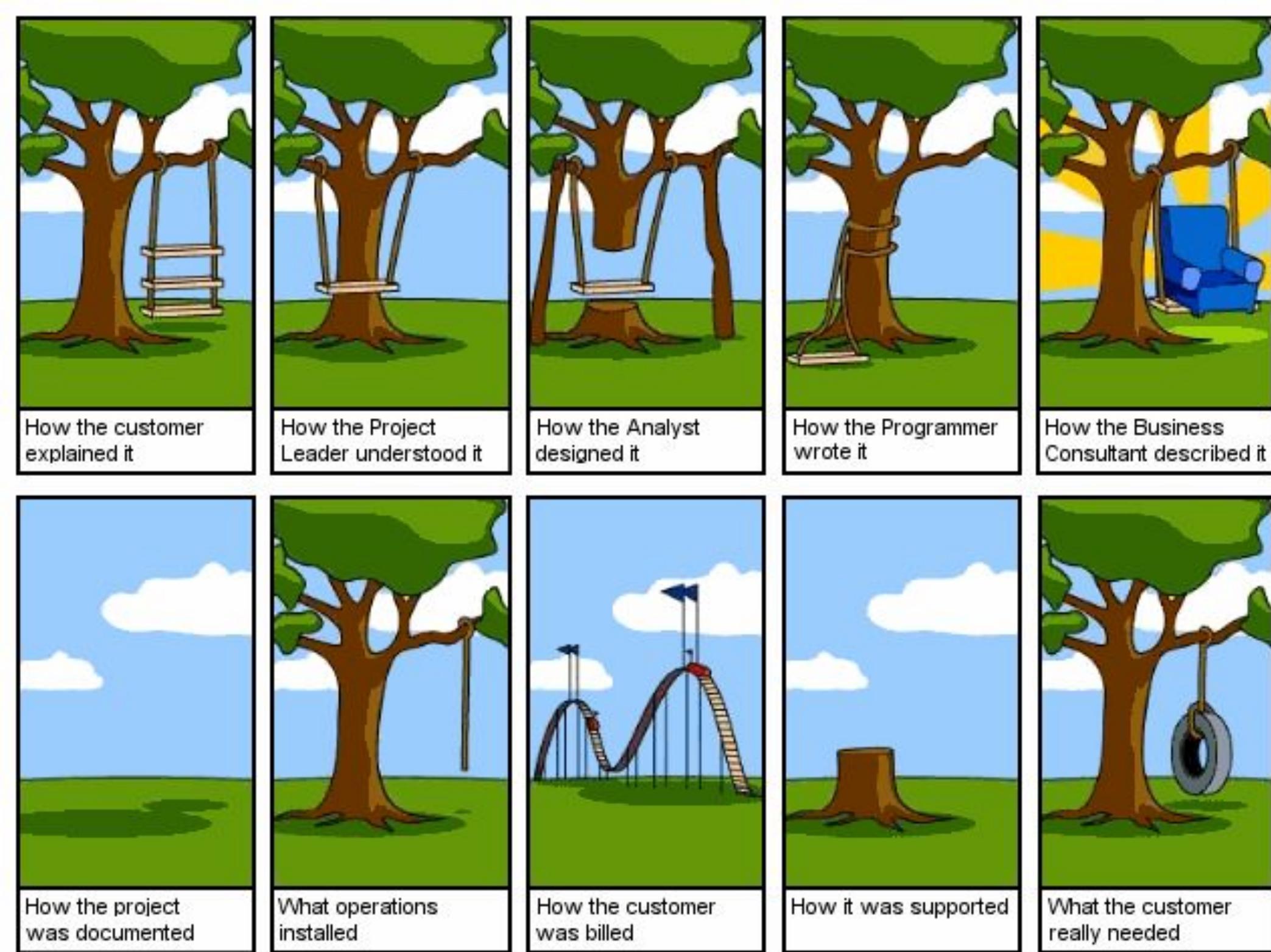
US DDX Submarine

- Viele Systeme
- Zusammen **30 Milliarde** Zeilen Code
- In **100+** Programmiersprachen
- Kosten: **\$15-30/Codezeile**



Woran liegt es?

- unzureichend spezifizierte Anforderungen
- häufiges Ändern der Anforderungen während des Projekts
- inkompetente Mitarbeiter
- fehlende Unterstützung durch das Management
- zu große Erwartungen
- falsche Schätzung der Zeit/Kosten
- Managementfehler





Rollen

- Der Entwickler/Architekt
 - schreibt den Code
- Fachberater
 - der die Spezifikation schreibt
- Tester
 - stellt die Qualität sicher
- Stakeholder
 - hat Interesse in dem Projekt/Produkt
- Endanwender
 - nutzt das Program



Stakeholder

Einzelpersonen und Organisationen

- die aktiv an einem Projekt beteiligt sind
- deren Interessen als Folge der Projektdurchführung oder des Projektabschlusses positiv oder negativ beeinflusst werden können
- die das Projekt und seine Ergebnisse beeinflussen



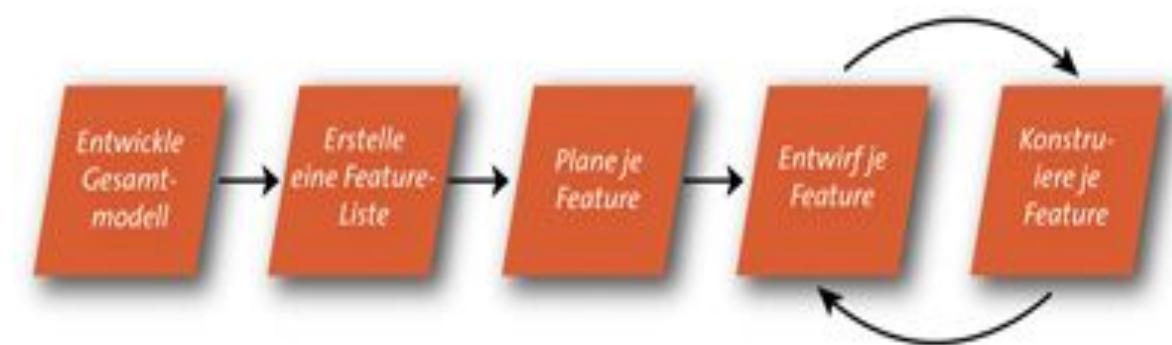
Softwaretechnik

Teilgebiet der Informatik

- Anforderungsanalyse
- Entwurf und Entwicklung von Software
- Organisation und Strukturierung der Entwicklung
- Projektmanagement
- Qualitätssicherung
- Betrieb und Wartung von Systemen

Softwaretechnik

- systematische **Verwendung** von
- **Prinzipien, Methoden und Werkzeugen**
- für die **Entwicklung** und Anwendung von umfangreichen **Softwaresystemen**





Softwaretechnik

- **Problemstellung (Idee)**
 - Eine Beschreibung eines Problems
 - Ein Lehrer braucht ein Programm für Studenten, die Rationale Zahlen lernen möchten.

- **Anforderungen (was?)**
 - Genaue Festlegung des Umfanges des geplanten Systems
 - beschreiben die Funktionalität des Produktes



Probleme

- Kunden wissen nicht was sie wirklich wollen
- Kunden benutzen ihre eigene Fachsprache
- Verschiedene Stakeholder können widersprüchliche Anforderungen haben
- Politische und organisatorische Faktoren können Anforderungen beeinflussen
- Anforderungen ändern sich während der Entwicklung
- Neue Stakeholder mischen sich ein



Anforderungen

- **Vollständig**
 - Alle Anforderungen des Kunden müssen explizit beschrieben sein
- **Atomar**
 - Es darf nur eine Anforderung pro Abschnitt beschrieben sein
- **Identifizierbar**
 - Jede Anforderung muss eindeutig identifizierbar sein
- **Nachprüfbar**
 - Die Anforderungen sollten mit Abnahmekriterien verknüpft werden, damit bei der Abnahme geprüft werden kann, ob die Anforderungen erfüllt wurden
- **Konsistent**
 - Die definierten Anforderungen sind untereinander widerspruchsfrei.



Features

- Features sind kleine Funktionen, die Wert für den Kunde haben
- Features werden nach dem einfachen Schema erstellt
<Aktion> <Ergebnis> <Objekt>
 - Aktion = eine Funktion, welche die Anwendung bereitstellt
 - Ergebnis = das Ergebnis der Ausführung der Funktion
 - Objekt = Die Entität, die die Funktion umsetzt
- Features können in wenigen Studen umgesetzt werden



Features

Problemstellung (Idee)

Ein Lehrer braucht ein Programm für Studenten, die Rationale Zahlen lernen möchten.

Rechner (Feature-Liste)

- F1. Eine Zahl einfügen
- F2. Rechner löschen
- F3. Die letzte Änderung rückgängig machen



Iteration

- Eine Iteration ist ein festgelegter Zeitraum innerhalb eines Projekts, in dem man eine stabile Version des Produkts zusammen mit Dokumentation erstellt
- Eine Iteration führt zu einem funktionierenden und nützlichen Programm für den Kunde
- Ablauf
 - man setzt in einer Iteration einige Funktionalitäten um
 - man zeigt die Ergebnisse (das Output) an
 - man bekommt Feedback



Feature Driven Development

- man muss eine Feature-Liste erstellen
- man muss die Iterations plannen
- Für jeden Iteration:
 - welche Features werden in der Iteration umgesetzt
 - Implementation und Testen
- Beispiel
 - Iteration 1
 - F1. Eine Zahl einfügen
 - F2. Rechner löschen
 - Iteration 2
 - F3. Die letzte Änderung rückgängig machen



Task Breakdown für Iteration 1 (Add, Löschen)

- zu Beginn einer Iteration muss man verstehen, was implementiert werden soll
- man muss dann die Features analysieren und die Arbeit in Aufgaben (Tasks) aufteilen
- T1. Berechnung den größten gemeinsamen Teiler
- T2. Addition zweier rationaler Zahlen
- T3. Rechner: Zahl einfügen
- T4 Rechner: Summe ausgeben
- T5. Rechner: Summe löschen
- T6. Benutzerschnittstelle. Menü für: Init, Add, Del
- **Aufgabenabhängigkeit** : T6 -> T5 -> T3 -> T2 -> T1



Testfall

- eine Reihe von Eingabewerten und erwarteten Ergebnisse, um eine bestimmte Funktion eines Programms zu testen

Input: a, b	c=ggT(a,b)
2 3	1
2 4	2
6 4	2
0 2	2
2 0	2
24 9	3
-2 0	Fehler
0 -2	Fehler



Labor 3+

- Man muss Konsoleanwendungen implementieren
- Code muss in Funktionen unterteilt werden
- Jede Funktion muss nur genau ein Ding tun
- Funktionen führen entweder Input/Output Operationen oder Berechnungen durch, aber nicht beides!
- Non-UI-Funktionen müssen spezifiziert werden



4. Ausnahmen und Module in Python





Inhalt

- Fehlerbehandlung
 - Spezifikationen
- Modulare Programmierung
- Wie schreibt man Module in Python
 - und warum



Fehlerhafte Programme

- Ein Programm kann aus vielen Gründen unerwünschtes Verhalten zeigen
- Fehler beim Entwurf
- Fehler bei der Programmierung des Entwurfs
 - Algorithmen falsch implementiert
- Umgang mit **außergewöhnlichen Situationen**
 - Abbruch der Netzwerkverbindung
 - Dateien können nicht gefunden werden
 - fehlerhafte Benutzereingaben



Umgang mit außergewöhnlichen Situationen

- Ausnahmesituationen unterscheiden sich von Programmierfehlern darin, dass man sie nicht (zumindest prinzipiell) von vornherein ausschließen kann
- Immer möglich sind zum Beispiel:
 - unerwartete oder ungültige Eingaben
 - Ein- und Ausgabe-Fehler beim Zugriff auf Dateien oder Netzwerk



Umgang mit außergewöhnlichen Situationen

- Die Erkennung und die Behandlung eines Fehlers muss oft in ganz verschiedenen Teilen des Programms stattfinden.
- Beispiel: Daten sind nicht konsistent (ein ID ist falsch)
 - Erkennung: Save Funktion
 - Behandlung: GUI



Ausnahmen

Eine **Ausnahme** (Exception) ist eine Ausnahmesituation, die sich während der Ausführung eines Programmes einstellt.

- Lässt man diese zu, so stürzt das Programm ab!
- Fängt man diese ab (Ausnahmebehandlung), läuft das Programm weiter!
- Die Auslösung einer Ausnahme bedeutet nicht automatisch, dass der Code einen Fehler enthält



Ausnahmen

Die meisten Programmiersprachen, die Ausnahmen unterstützen, verwenden eine gemeinsame Terminologie und Syntax

- Ausnahmen auslösen
- Ausnahmen abfangen oder behandeln
- Verbreitung
- try / raise (throw) und except (catch) Keywords



Ausnahmebehandlung

- Ausnahmebehandlung (Exception handling)
 - der Prozess, bei dem Fehlerzustände in einem Programm systematisch behandelt werden

try:

```
#Code der Ausnahmen auslösen kann  
except <ErrorType>:  
    #Code der die Situation beherrscht
```

- was hinter try kommt, wird ausgeführt, bis ein Fehler auftritt
- was hinter except kommt, wird nur ausgeführt, wenn im try-statement eine Exception der angegebenen Art aufgetreten ist
- Ja! Man muss die verschiedenen ErrorTypes wissen



Beispiel

- wir haben den folgenden Code

```
def div(a,b):  
    return a / b  
  
print(div(1,2))  
print(div(0,1))  
print(div(1,0))
```

```
0.5  
0.0  
Traceback (most recent call last):  
  File "main.py", line 6, in <module>  
    print(div(1,0))  
  File "main.py", line 2, in div  
    return a / b  
ZeroDivisionError: division by zero  
▶ █
```

- wir wollen diese Situation verhindern
- wir können if verwenden
 - aber in dem Fall muss man extra Logik in der Funktion reinstecken



Beispiel

- Eleangeter geht es mit Exceptions

```
def div(a,b):  
    try:  
        r = a / b  
    except ZeroDivisionError:  
        r = None  
    return r  
  
print(div(1,2))  
print(div(0,1))  
print(div(1,0))
```

```
0.5  
0.0  
None  
▶ □
```



None, NoneType, Pass

- **None** ist ein Objekt ohne Wert
- **NoneType** ist der Typ dieses Objektes
- **None** als Rückgabewert zeigt das
 - die Funktion returniert nichts
 - zB eine Suchfunktion hat nichts gefunden
- **pass** ist eine Anweisung, die kein Ergebnis hat
 - nützlich, um Code zu strukturieren
 - kann verwendet werden, um zu zeigen
 - das code wird dort irgendwann geschrieben

```
def add(a,b):  
    pass
```



Python Syntax

- Wenn man Ausnahmen abfangen will, muss der Code in einem `try-except` Block enthalten sein
- Ausnahmen werden anhand ihres Typs abgefangen
- Ein `try`-Block kann **einen**, **mehrere** oder **alle** Ausnahmetypen abfangen
- Das Erstellen von Ausnahmen in unserem Code erfolgt mit dem Schlüsselwort `raise`
- Man kann zusätzliche Argumente (zB eine Fehlermeldung) für jede Ausnahme, die ausgelöst wird, bereitstellen



Ausnahmebehandlung

- Eine Ausnahme kann behandelt werden durch:
 - Die Funktion, bei der die Ausnahme ausgelöst wurde
 - Jede Funktion, die **dieser** Funktion aufruft
 - der Python-runtime - dies wird zu einem Abbruch des Programmes führen
- der Satz "unhandled exception has occurred in your application..." muss uns bekannt sein
- jetzt können wir verstehen, was dort passiert wurde!



Ausnahmebehandlung. Wann?

- Um eine Ausnahmesituation zu signalisieren - die Funktion kann den Vertrag nicht erfüllen
 - z. B. Vorbedingungen sind nicht erfüllt
 - oder eine Situation aufgetreten wurde, in der die Funktion nicht weitergehen kann
 - eine erforderliche Datei wurde nicht gefunden
 - ist nicht zugreifbar
 - usw.
- Um Vorbedingungen durchzusetzen
- Generell sollte man keine Ausnahmen verwenden, um den Programmfluss zu steuern!



Ausnahmebehandlung

- Wie kann man überprüfen, ob ein Parameter einer Funktion von einem bestimmten Typ ist?

- mit `type()/isinstance()`

```
def add(a,b):  
    if type(a) == int and typ(b) == int:  
        # isinstance(a,int)  
    ...
```

- mit Exceptions

- `TypeError/AttributeError`

- Type Annotations

```
def do_something (n:int)
```

- Hinweis: kann man aber muss man nicht

- kann ein Indikator für schlechtes Design sein
 - die Spezifikation einer Funktion ist wichtiger

```
1  def my_max(arg):  
2      try:  
3          return max(arg)  
4      except TypeError:  
5          return 0  
6  
7  
8  print(my_max([1,2,3]))  
9  print(my_max(3))  
10
```



Types

- **SyntaxError** – es ist ein syntaktischer Fehler im Quelltext
- **IOError** – eine Datei existiert nicht, man darf nicht schreiben, die Platte ist voll
- **IndexError** – in einer Sequenz gibt es das angeforderte Element nicht
- **KeyError** – ein Mapping hat den angeforderten Schlüssel nicht
- **ValueError** – eine Operation kann mit diesem Wert nicht durchgeführt werden



Modulare Programmierung

- eine Softwaretechnik, die das Ausmaß erhöht, in dem Software aus unabhängigen, austauschbaren Komponenten besteht
 - Jede solche Komponente erfüllt einen Aspekt innerhalb des Programms und enthält alles, was dazu erforderlich ist.
 - Module in Python
- Module sind daher
 - Unabhängig
 - Austauschbar
- Ermöglichen die Gruppierung von Funktionen
- Ermöglichen die einfachere Bereitstellung von Funktionen
- Helfen bei der Lösung von Namenskonflikten



Modulare Programmierung in Python

- ein Python Modul ist eine Datei die aus Anweisungen und Definitionen besteht
- **Name**: der Dateiname ist der Modulname mit .py ergänzt
- **Docstring**
 - Drei öffnende Anführungszeichen
 - eine kurze Beschreibung in einem Satz
 - eine Leerzeile
 - weitere Bemerkungen
 - abschließend drei Anführungszeichen in einer eigenen Zeile
- **Anweisungen**
 - ein Modul kann ausführbare Anweisungen wie auch Funktionsdefinitionen enthalten
 - diese Anweisungen dienen der Initialisierung des Moduls
 - **keine “globale” Variablen/Namen**
 - sie werden nur dann ausgeführt, wenn das Modul das erste mal importiert wird



die import-Anweisung

- Um ein Modul verwenden zu können, muss es zuerst importiert werden.
- Die Importanweisung:
 - a. Durchsucht den globalen Namespace nach dem Modul. Wenn das Modul existiert, ist es bereits importiert und man muss nichts weiter machen
 - b. Sucht nach dem Modul.
 - c. Im Modul definierte Variablen und Funktionen werden in eine neue Symboltabelle (einen neuen Namespace) eingefügt. Nur der Modulname wird zur aktuellen Symboltabelle hinzugefügt

```
Python 3.8.2 (default, Feb 26 2020, 02:56:10)
> sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
> from math import sqrt
> sqrt(2)
1.4142135623730951
> █
```



die import-Anweisung

```
from math import sqrt  
sqrt(2)
```

```
import math  
math.sqrt(2)
```

```
from math import *  
sqrt(2)
```



Beispiel

Modul
`useful_functions.py`

mit folgenden Funktionen

- `add(a,b)`
- `mul(a,b)`
- `sub(a,b)`



der Modul-Suchpfad

```
import spam
```

Wenn das Modul *spam* importiert wird, sucht der Interpreter nach einer Datei mit Namen *spam.py*

- im aktuellen Verzeichnis
 - wo das Skript gespeichert ist
- in der Liste der Verzeichnisse, die durch die Umgebungsvariable **PYTHONPATH** spezifiziert wird
- in der Liste der Verzeichnisse, die durch die Umgebungsvariable **PYTHONHOME** spezifiziert wird.
 - Abhangig von der Installation
 - /usr/local/lib/python (Unix)
- wenn das Modul nicht gefunden wurde, wird die **ImportError** Ausnahme ausgelöst



Packages

- Packages sind eine Möglichkeit, um Modulen zu strukturieren
- A.B zeigt das B ein Modul in Package A ist
 - Auf dem Laufwerk stellen Folders Packages dar
 - B.py befindet sich in einem Folder A.
- Für den Import von Paketen gelten die gleichen Regeln wie für Module
- Jeder Folder, der ein Package darstellt, enthält eine `__init__.py` Datei
- `__init__.py` kann leer sein oder Initialisierungscode enthalten.



Labor 3+

Man muss Module für folgende erstellen:

- Benutzeroberfläche
 - Funktionen für die Benutzerinteraktion.
 - Enthält Eingabe- und Ausgabefunktionen sowie Funktionen für Datenüberprüfung
- Businesslogik
 - Enthält Funktionen, die zum Implementieren von Programmfunctionen erforderlich sind



Beispiel

Lass un Python-Programm schreiben, welches eine Liste von Studenten verwaltet. Jeder Student hat als Attribute Name, Universität.

Funktionalität

- Studenten anlegen und finden
- Studentinfo ausgeben

Übung



Neue Funktionalität

Sort: Studenten nach Name sortieren



Benutzerdefinierte Typen I

{} () . - " " . ; , \
{ } () . - " " . ; , \
V V , , = ; t / =
` ` , , , , ,
/ / | | | | | |
/ ,)) | ,))



- Benutzerdefinierte Typen
- Klassen und Objekte
- Erstellung\Verwendung von benutzerdefinierten Typen bzw. Klassen in Python





Klassen und Objekte

Objektorientierte Programmierung (OOP) ist eine Methode zur
Modularisierung von Programmen

Die Basis von allem:

das Objekt (= Daten + Funktionalität)



Objekte

- beschreiben einen Gegenstand, Person etc. aus der realen Welt
- haben Felder/Attribute (Eigenschaften), die das Aussehen des Objekts beschreiben
- haben Methoden, die die Attribute verändern
- *sind Substantive in einem Text*



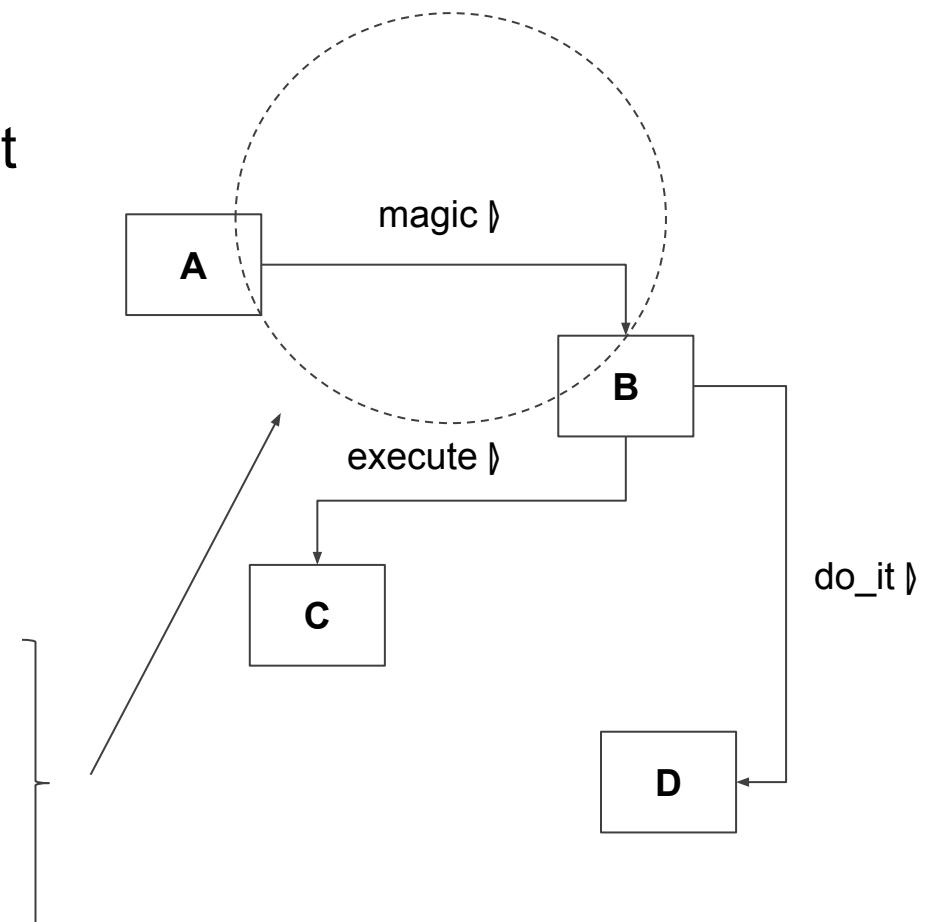
Beispiel

Schreibe eine Anwendung, welche das Spiel Hangman für die Konsole implementiert.

Es soll von einem **Spieler** beliebig oft spielbar sein. In jedem Schritt des **Spiels** soll die verbleibende Menge an Rateversuchen, sowie die bereits erratenen Buchstaben des Lösungswortes angezeigt werden.

Programme

- es gibt kein globaler Zustand
- der Zustand des Programms ist durch die Zustände aller Objekte beschrieben
- Objekte kommunizieren miteinander
- Beispiel:
 - Objekt **B** hat die Methode `magic`
 - Objekt **A** ruft `magic` auf (message passing)







just Code...

```
while True:  
    print ("""  
    1 - add  
    2 - mul  
    ...  
    """"  
)  
opt = int(input("select?"))  
  
if opt == 1:  
    a = int (input("a="))  
    b = int (input("b="))  
    number1 = (a,b)  
  
    a = int (input("a="))  
    b = int (input("b="))  
    number2 = (a,b)  
  
    rez = number[1]+...  
if opt == 2:  
    a = int (input("a="))  
    b = int (input("b="))  
    number1 = (a,b)  
  
    a = int (input("a="))  
    b = int (input("b="))  
    number2 = (a,b)  
  
    rez = number[1]+...
```



Prozedurale Programmierung

```
def addition (r, total):
    return rational(r.a*total.b + total.a+r.b,r.b*total.b)

def multiplication(r, total):
    ...

def menu():
    return """
        1 - add
        ...
    """
def main():
    total = 0
    while True:
        print (menu())
        opt = int(input("select?"))
        ...
    
```



Modulare Programmierung

start.py

```
def menu():
    return """
        1 - add
        ...
        """
def main():
    while True:
        print (menu())
        opt = int(input("select?"))
        ...
```

rationaloperationen.py

```
def add (r1, r2):
    return rational(r1.a*r2.b +
r2.a+r1.b,r1.b*r2.b)

def mult (r1,r2):
    return ...

def to_string(r):
    return "%i/%i" %(r.a,r.b)
```

GUI (**start.py**)Rechner (**rechneroperationen.py**)Rational (**rationaloperationen.py**)

rechneroperationen.py

```
def addition (r, total):
    ...
    def multiplication(r, total):
        ...
```



Modulare Programmierung

- Eine Anforderung wird in vielen kleinen Aufgaben/Tasks zerlegt
- Jede kleine Aufgabe/Task ist in sich abgeschlossen
- Jede kleine Aufgabe/Task ist in einer Datei abgelegt

classes

just. add. water.





OOP

- abstrahiert Gegenstände der realen Welt
- beschreibt und verändert Objekte
- versucht Daten und Funktionen eines Objekts in einer Struktur zu kapseln.
 - Die Daten beschreiben das Objekt.
 - Funktionen verändern die Attributwerte eines Objekts.



Prinzipien der OOP

Abstraktion

- Objekte der realen Welt werden nachgebildet

Datenkapselung

- das Objekt ist eine Black Box
- In dieser Black Box wird mit Hilfe von Felder das Objekt beschrieben
- Funktionen verändern die Felder eines Objekts



Beispiel

Ein Würfel

- Der Würfel ist rot -> Farbe
- Der Würfel hat eine Kantenlänge von 5 cm -> Länge
- *Der Würfel kann gedreht werden.*
- *Der Würfel kann neu eingefärbt werden.*



Beispiel

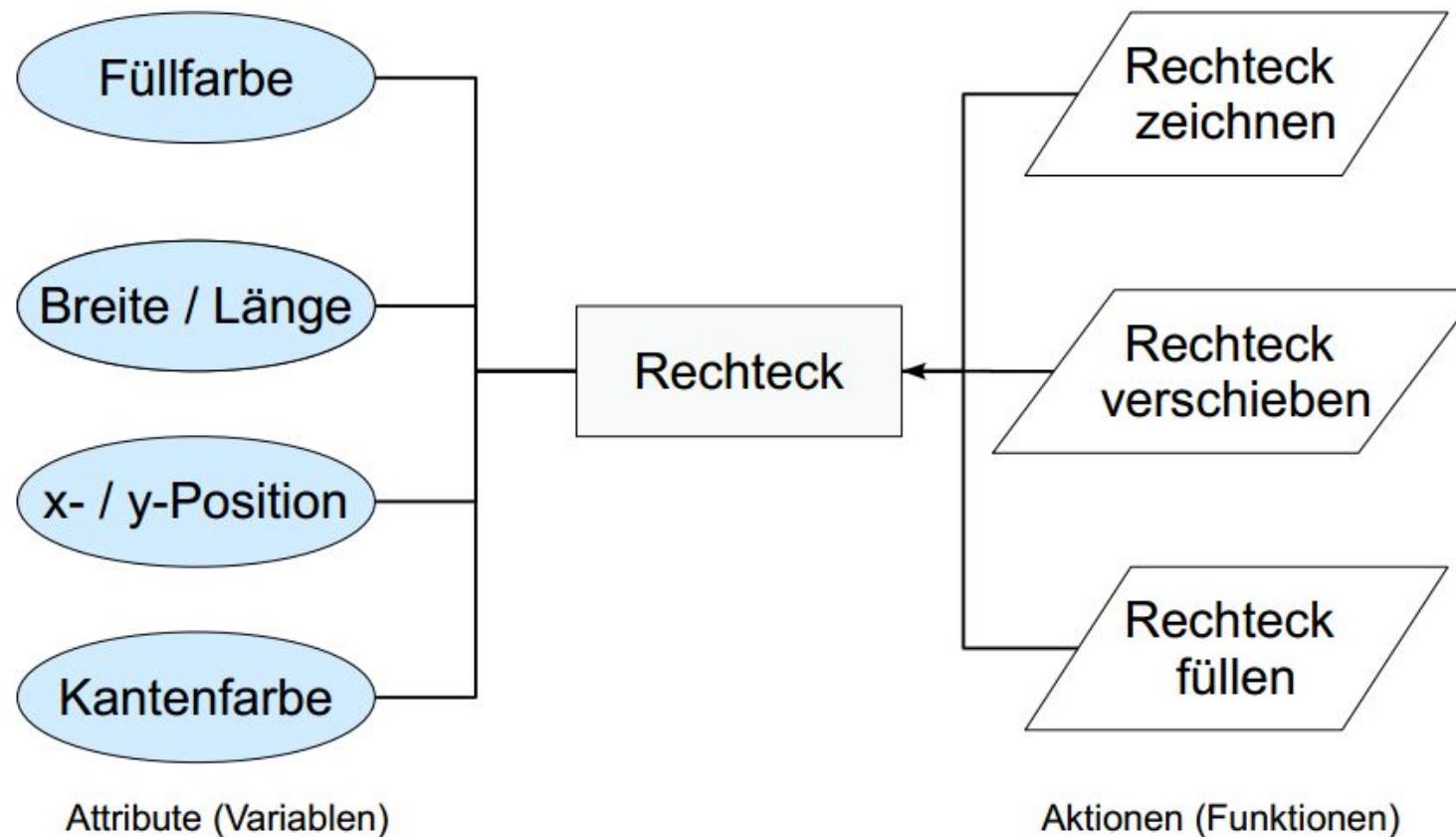
Welche Eigenschaften hat jeder Würfel?

- Farbe
- Kantenlänge

Welche Methoden hat jeder Würfel?

- Drehen
- Einfärben
- Zeichnen

Beispiel





Python

Everything is a object. In Python ist jedes Element ein Objekt.

Objekte:

- können Felder und Funktionen haben
- sind an einen bestimmten Datentyp gebunden
- können an Funktionen übergeben werden
- werden mit Hilfe von Klassen beschrieben
- werden mit speziellen Methoden erzeugt und initialisiert

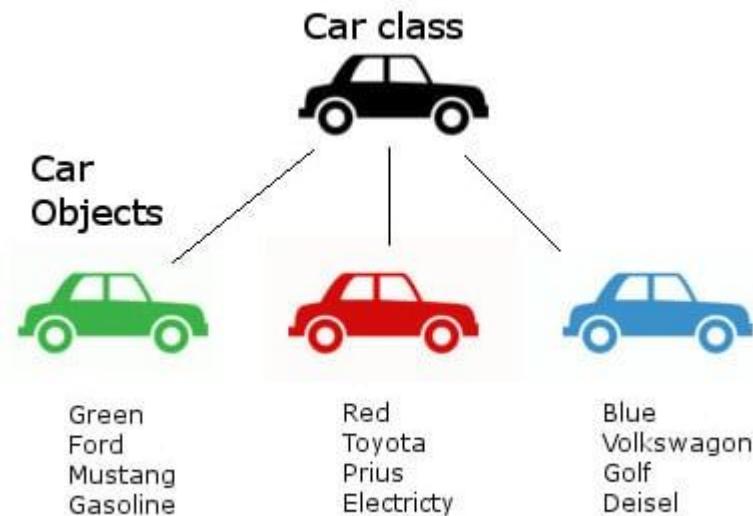


Beispiel

```
>>> #integer  
>>> x = 1  
>>> x.__add__(2) # x = 1 + 2  
>>> 3  
  
>>>  
>>> #Listen  
>>> l = [1, 2]  
>>> l.__add__([2]) # l + [2]  
[1, 2, 2]  
>>> l  
[1, 2]  
>>>
```

Klassen

ein abstraktes Modell bzw. ein Bauplan für eine Reihe von ähnlichen Objekten



beschreiben Attribute (**Eigenschaften**) und Methoden (**Verhaltensweisen**) der Objekte.



Klassen

Definition:

- wird mit dem reservierten Wort `class` eingeleitet,
- danach kommt der Name der neuen Klasse,
- ein Doppelpunkt und wieder ein compound statement (**Einrückung!**)

```
class MyClass:  
    <statement 1>  
    ....  
    <statement n>
```



UML Beschreibung

- **Unified Modelling Language** zur Darstellung von Klassen
- Der Name der Klasse steht am oberen Rand
- Dem Namen folgen die Attribute der Klasse und darunter die Methoden
- In UML werden private Methoden und Attribute mit einem Minuszeichen und öffentliche Attribute und Methoden mit einem Pluszeichen gekennzeichnet.

Wuerfel
- farbe : string - kante: double
+get_Farbe() : string +get_Laenge() : float +set_Farbe() : string +set_Laenge(): float +drehen_Wuerfel() : void



Konstructor

Konstruktor: eine Methode, die beim Erzeugen eines Objekts dieser Klasse aufgerufen wird.

```
x = MyClass()
```

Jedes Objekt hat einen eigenen Namespace. Namen darin heißen Attribute des Objekts.

```
class MyClass:  
    def __init__(self):  
        self.someData = []
```



Destruktor

Mit Hilfe von del RechteckBlau
wird automatisch der dazugehörige Destruktor
def __del__() aufgerufen

Die Methode wird implementiert, wenn zum Beispiel ...

- Netzwerkverbindungen getrennt werden müssen
- Dateien geschlossen werden müssen
- bestimmte Fehler abgefangen werden



Eine Instanz

```
class Student:  
    def __init__(self, name):  
        self.name = name  
  
bob = Student("bob")
```

(Name des Objekts) (Name der Klasse)

- erweist auf ein bestimmtes Objekt einer bestimmten Kategorie
- ist ein Synonym für ein Objekt
- die Parameter sind im Konstruktor definiert



Felder/Attribute

- beschreiben den Zustand eines Objekts - Gegenstand, Person etc
- Jedes Objekt einer Klasse hat die gleiche Attribute
- Jedes Objekt einer Klasse unterscheidet sich aber in mindestens einem Attributwert von allen anderen Objekten



Felder/Attribute

```
self.n = a  
n = a
```

```
class RationalNumber:  
    """  
        Abstract data type for rational numbers  
        Domain: {a/b where a and b are integer numbers b!=0}  
    """  
  
    def __init__(self, a, b):  
        """  
            Creates a new instance of RationalNumber  
        """  
        self.n = a  
        self.m = b  
  
r1 = RationalNumber(1,3) #create the rational number 1/3
```



Self I

- ist ein Platzhalter für den Aufrufer der Methode
- beantwortet die Frage „Wer hat die Methode aufgerufen?“
- ist meist das erste Argument einer Methode
- beschreibt die Instanz, die die Methode aufgerufen hat.



Methoden

- beschreiben das Verhalten eines Objekts
- lesen oder verändern Attributwerte
- beschreiben eine Schnittstelle nach außen
- werden innerhalb der Klasse definiert
- werden nur einmal für die Klasse im Speicher angelegt
- ergeben sich aus den Attributen und deren Nutzung in einer Klasse

Methoden



```

def testCreate():
    """
        Test function for creating rational numbers
    """

    r1 = RationalNumber(1,3) #create the rational number 1/3
    assert r1.getNominator()==1
    assert r1.getDenominator()==3
    r1 = RationalNumber(4,3) #create the rational number 4/3
    assert r1.getNominator()==4
    assert r1.getDenominator()==3

class RationalNumber:
    """
        Abstract data type rational numbers
        Domain: {a/b where a,b integer numbers, b!=0, greatest common divisor
        a, b =1}
    """

    def __init__(self, a, b):
        """
            Initialize a rational number
            a,b integer numbers
        """
        self.__nr = [a, b]

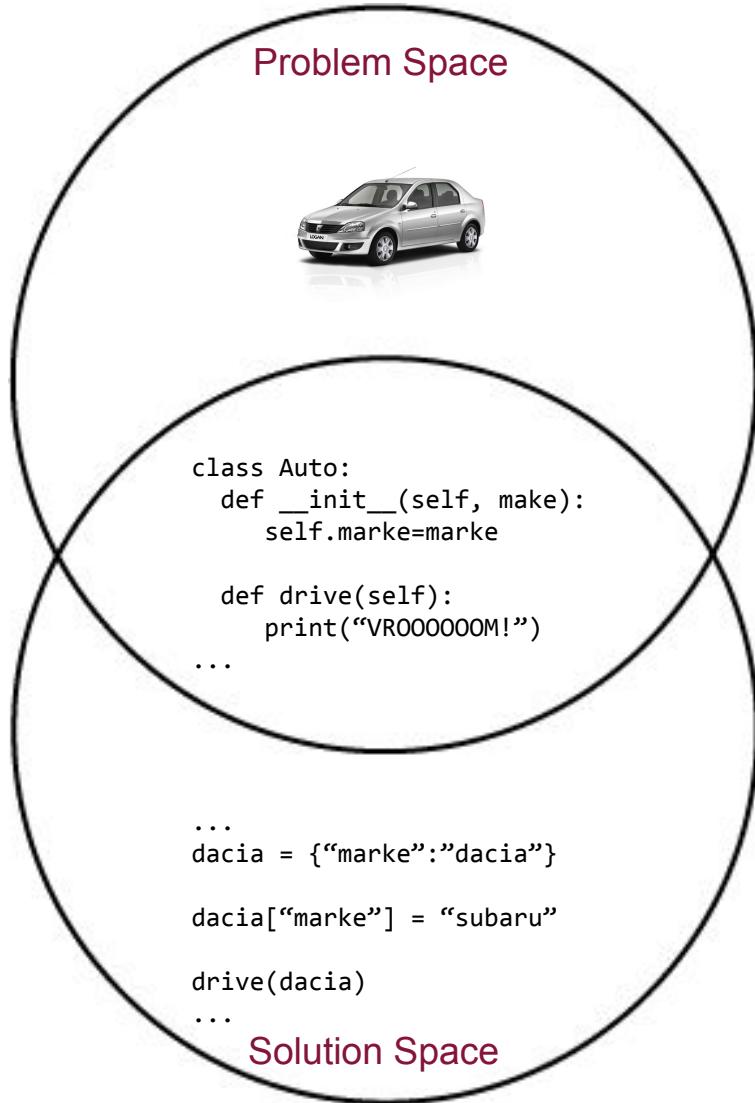
    def getDenominator(self):
        """
            Getter method
            return the denominator of the rational number
        """
        return self.__nr[1]

    def getNominator(self):
        """
            Getter method
            return the nominator of the method
        """
        return self.__nr[0]

```

- Die Methode wird mit der Instanz immer durch ein Punkt verbunden.
- Falls die Methode nicht definiert ist, wird die Fehlermeldung „**AttributeError**“ ausgegeben.

Beispiele...Autos und Students



Beispiel

```
class Auto:
```

```
    def __init__(self):
```

```
        self.kilometerstand = 0
```

```
    def drive(self):
```

```
        self.kilometerstand += 1
```

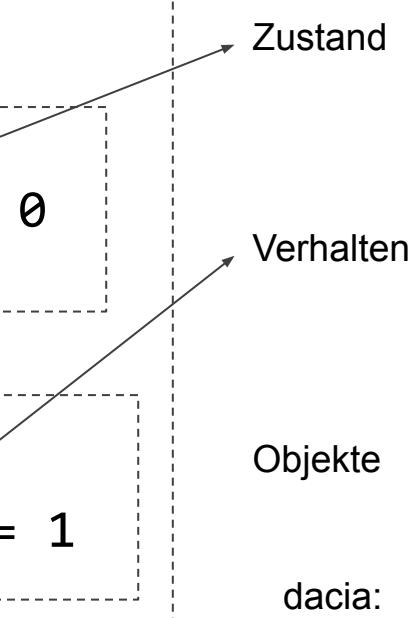
```
dacia = Auto()
```

```
lada = Auto()
```

```
dacia.drive()
```

```
dacia.drive()
```

```
lada.drive()
```



dacia:

kilometerstand

2



lada:

kilometerstand

1





Self II

ist ein Platzhalter für den Aufrufer der Methode

```
class Auto:  
    def __init__(self, farbe):  
        self.kilometerstand = 0  
        self.farbe = farbe  
  
    def drive(self, km):  
        self.kilometerstand += km  
  
dacia = Auto("rot")  
lada = Auto("blau")
```

dacia.drive(10)
lada.drive(200) ↔ drive(dacia, 10)
 ↔ drive(lada, 200)



Beispiel

- Implementiere eine Klasse **Auto**
- Jedes Auto hat für Felder/Attribute:
 - Marke als String
 - Modell als String
 - Farbe als String
 - Baujahr als Integer
- Implementiere eine Klasse **Statistics**
- die Klasse soll für eine Reihe von Autos berechnen:
 - die Anzahl von Autos mit einer eingegebenen Farbe
 - das durchschnittliche Baujahr für alle Autos einer Marke

Tests

Spezifikation



Benutzerdefinierte Typen II

{}) . - " " . A . - . A
{ (\ V \ , ; , \ = ; t / =
} \] " . ' , -- ' .
/ / [] [])
/ ,) [,))



magic operator

```
def s (a,b):  
    return a+b
```

```
def p (a,b):  
    return a*b
```

```
def op (a,b,mop):  
    r = mop(a,b)  
    return r
```

```
print(op(1,2,s)) #-> 3  
print(op(2,3,p)) #-> 6
```



magic operator

```
def par (t):  
    return t%2 == 0  
  
def filter (l, op):  
    ll = []  
    for el in l:  
        if op(el):  
            ll.append(el)  
    return ll  
  
print(filter([1,2,3,4], par))
```



Lambdas

- Lambda-Funktionen kommen aus der funktionalen Programmierung
- Mit Hilfe des `lambda`-Operators können **anonyme Funktionen**, d.h. Funktionen ohne Namen erzeugt werden

`lambda` Argumentenliste: Ausdruck

```
s = lambda x, y : x + y  
s(1,2) #3
```



map

```
r = map(func, seq)
```

- `func` ist eine Funktion und `seq` eine Sequenz (z.B. eine Liste)
- `map` wendet die Funktion `func` auf alle Elemente von `seq` an und schreibt die Ergebnisse in eine neue Liste

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

```
b = [4,5,6]
```

```
def mal2(el): return el*2
```

```
list(map(lambda el: el*2, a)) #[2,4,6]
```

```
list(map(mal2, a)) #[2,4,6]
```

```
list(map(lambda ela,elb: ela+elb, a, b)) #[5,7,9]
```



filter

```
filter(funktion, liste)
```

bietet eine elegante Möglichkeit diejenigen Elemente aus der Liste liste herauszufiltern, für die die Funktion funktion True liefert

```
a = [1,2,3,4]
```

```
list(filter(lambda el:el%2 == 0, a)) #[2,4]
```



reduce

```
r = reduce(func, seq)
```

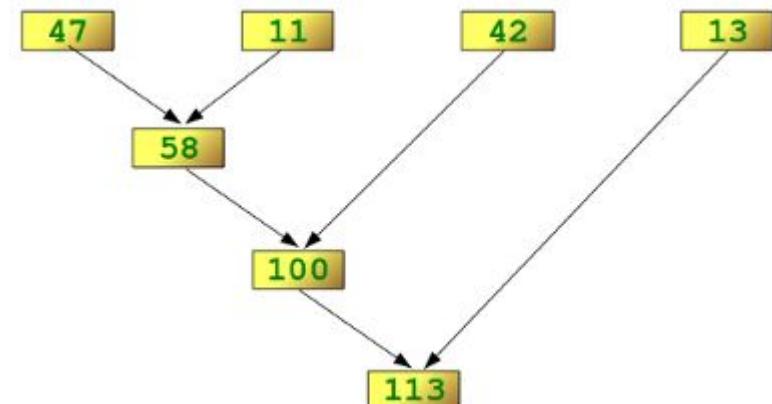
- wendet die Funktion func fortlaufend auf eine Sequenz an und liefert einen einzelnen Wert zurück.
- zuerst wird func auf die beiden ersten Argumente s_1 und s_2 angewendet.
- das Ergebnis ersetzt die beiden Elemente s_1 und s_2 :

$$[func(s_1, s_2), s_3, \dots, s_n]$$

- im nächsten Schritt wird func auf $func(s_1, s_2)$ und s_3 angewendet.
- dies wird solange fortgesetzt bis nur noch ein Element übrig bleibt

```
import functools
```

```
functools.reduce(  
    lambda x,y: x+y, [47,11,42,13]  
) #113
```





List Comprehension

eine elegante Methode Listen in Python zu definieren oder zu erzeugen

```
l = [1,2,3,4,5]
[el*2 for el in l] #wie map
[el for el in l if el%2 == 0] #wie filter

[(a,b,c) for a in range(1,30) for b in range(a,30) for c
in range(b,30) if a**2 + b**2 == c**2]
# die pythagoreischen Tripel
```



List Comprehension

```
l = [0]*5
```

```
a = [1,2,3,5]
```

```
b = [1,4,3,4]
```

```
s = [i+j for i in a for j in b]
```

```
l = [i for i in range(len(a)) for j in range(len(b)) if a[i] == b[j]]
```

```
m = [ [0 for i in range(5)] for j in range(5)]
```

```
m = [[0] * 5] * 5 #problematisch
```

```
m = [[1 if i == j else 0 for i in range(5)] for j in range(5)]
```



Anwendung: Autos

- Ein Autohaus hat Autos zu verkaufen
- Kunden haben Geld
- Kunden können vom Autohaus Autos kaufen



Anwendung: Autos

- Ein Autohaus hat Autos zu verkaufen
 - Auto:
 - Attribute: Modell, Farbe, Baujahr
 - Methoden: tanken, anlassen
- Kunden haben Geld
- Kunden können vom Autohaus Autos kaufen



Anwendung: Autos

- Ein Autohaus hat Autos zu verkaufen
 - Auto:
 - Attribute: Modell, Farbe, Baujahr
 - Methoden: tanken, anlassen
- Kunden haben Geld
 - Kunde:
 - Name, Betrag
 - Methoden: verdienen
- Kunden können vom Autohaus Autos kaufen



Anwendung: Autos

- Ein Autohaus hat Autos zu verkaufen
 - Auto:
 - Attribute: Modell, Farbe, Baujahr
 - Methoden: tanken, anlassen
- Kunden haben Geld
 - Kunde:
 - Name, Betrag
 - Methoden: verdienen
- Kunden können vom Autohaus Autos kaufen
 - Autohaus:
 - Autos, Sold
 - Methoden: add_auto, verkaufen



Slots

- Jedes Python-Objekt hat ein Attribut `__dict__`
 - das ein Dict ist und alle anderen Attribute der Klasse enthält
- z.B. für `self.attr` macht Python tatsächlich
 - `self.__dict__['attr']`

```
class T:  
    def __init__(self):  
        self.t = 0  
  
t = T()  
print(t.t)  
print(t.x) #fehler  
t.x=10  
print(t.x) #ok
```



Slots

- `__slots__` wenn wir viele (Hunderte, Tausende) Objekte derselben Klasse instanziieren möchten
- `__slots__` gibt es nur als Tool zur Speicheroptimierung
- `__slots__` soll nicht zur Einschränkung der Attribut-Erstellung verwendet sein

```
class T:  
    __slots__ = 'a', 'b'  
  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b
```



Öffentliche Attribute

Klasse.Attribut = <Wert>

können von außen zugegriffen werden

als Klassenvariable: `klasse.variable`

als Objektvariable: `objekt.variable`

Klassenvariablen sind standardmäßig öffentlich



Private Attribute?

Klasse.__Attribut = <Wert>

- beginnen mit zwei Unterstrichen
- ein Zugriff von außen auf diese Attribute ist theoretisch nicht möglich*.
- werden mit Hilfe von get-Methoden zugegriffen.
- werden mit Hilfe von set-Methoden verändert.

***results may vary... (name mangling)**



Private Attribute?

Get/Set-Methoden

```
def getPosY(self):  
    return self.__yPos
```

```
def setPosY(self, pos):  
    self.__yPos = pos
```



Private Attribute?

```
class T:  
    def __init__(self, a):  
        self.__a = a
```

```
t = T(10)  
t.__a = 10 #error  
t._T__a = 10
```



Schwache private Attribute

`Klasse._Attribut = <Wert>`

- beginnen mit einem Unterstrich
- nur als Info für den Aufrufer
- können von außen zugegriffen werden
- werden nicht durch die Anweisung
`from ... import *` in eine Datei importiert



The Python way

```
class T:  
    def __init__(self,x):  
        self.__x = x  
  
    @property  
    def x(self):  
        return self.__x  
  
    @x.setter  
    def x(self, x):  
        self.__x = x
```



Klassenvariablen

Eine Klassenvariable kann nur mit Hilfe des Klassennamens verändert werden (theoretisch)

Der Klassenname und das Attribut werden durch einen Punkt miteinander verbunden

Modul.Klasse.Attribut = Wert



Klassenvariablen

- werden innerhalb der Klasse, aber außerhalb einer Methode definiert
- werden häufig zu Beginn des Klassenrumpfes aufgelistet
- sind Attribute, die alle Objekte besitzen
- **können von jedem Objekt der Klasse verändert werden**
- sind globale Attribute eines Objekts



Klassenvariablen

```
class Rechteck(object):
    ANZAHL = 0
    FARBE_KANTE = "Black"
    FARBE_FÜLLUNG = "White"

    def __init__(self, b = 10, h = 10):
        self.__xPos = 0
        self.__yPos = 0
        self.hoehe = h
        self.breite = b
        Rechteck.ANZAHL = Rechteck.ANZAHL + 1
```



Benutzerdefinierte Typen III

```
t : List( Unit | List( t, t ) )
```



- Statische Elemente in Python
- Operatoren in Python
- Code Organisation / Design Prinzipien



Zwischenprüfung - 09.12

SEMINAR: $2+2=4$



KLAUSUR: DANIEL HAT EINEN APFEL. BERECHNE DIE MASSE DER SONNE



Beispiel

```
class Konto(object):

    def __init__(self, inhaber, kontonummer,
                 kontostand,
                 kontokorrent=0):
        self.Inhaber = inhaber
        self.Kontonummer = kontonummer
        self.Kontostand = kontostand
        self.Kontokorrent = kontokorrent

    def ueberweisen(self, ziel, betrag):
        if(self.Kontostand - betrag < -self.Kontokorrent):
            # Deckung nicht genuegend
            return False
        else:
            self.Kontostand -= betrag
            ziel.Kontostand += betrag
            return True

    def einzahlen(self, betrag):
        self.Kontostand += betrag

    def auszahlen(self, betrag):
        self.Kontostand -= betrag

    def kontostand(self):
        return self.Kontostand
```

```
>>> from konto import Konto
>>> K1 = Konto("Jens", 70711, 2022.17)
>>> K2 = Konto("Uta", 70813, 879.09)
>>> K1.kontostand()
2022.17
>>> K1.ueberweisen(K2, 998.32)
True
>>> K1.kontostand()
1023.85
>>> K2.kontostand()
1877.41
```



Klassenvariablen

Eine Klassenvariable kann nur mit Hilfe des Klassennamens verändert werden (theoretisch)

Der Klassenname und das Attribut werden durch einen Punkt miteinander verbunden

Modul.Klasse.Attribut = Wert



Klassenvariablen

- werden innerhalb der Klasse, aber außerhalb einer Methode definiert
- werden häufig zu Beginn des Klassenrumpfes aufgelistet
- sind Attribute, die alle Objekte teilen
- **können von jedem Objekt der Klasse verändert werden**
- sind globale Attribute eines Objekts



Klassenvariablen

```
class Rechteck:  
    ANZAHL = 0  
    FARBE_KANTE = "Black"  
    FARBE_FÜLLUNG = "White"  
  
    def __init__(self, b = 10, h = 10):  
        self.__xPos = 0  
        self.__yPos = 0  
        self.hoehe = h  
        self.breite = b  
        Rechteck.ANZAHL = Rechteck.ANZAHL + 1
```



weiteres Beispiel

```
class RationalNumber:  
    numberofInstances = 0  
    def __init__(self,a,b):  
        self.n = a  
        self.m = b  
        RationalNumber.numberofInstances += 1  
  
def testNumberInstances():  
    assert RationalNumber.numberofInstances == 0  
    r1 = RationalNumber(1,3)  
    assert r1.numberofInstances == 1  
    r1.numberofInstances = 8  
    assert r1.numberofInstances == 1  
testNumberInstances()
```



weiteres Beispiel

```
1  class T:
2      t = 0
3      def __init__(self, x):
4          self.x = x
5          T.t += 1
6          self.x += 1
7
8
9      T.t = 1000
10
11     print('here')
12     assert 1 == 1
13
14     t = T(10)
15     print('t.x=', t.x)
16
17
18
19     t1 = T(10)
20     print(T.t)
21
22     t.t = 10
23     #T.t = 101
24
25     print(T.t, t.t, t1.t)
```



Statische Methoden

```
class T:  
    __counter = 0  
  
    def __init__(self):  
        type(self).__counter += 1  
  
    @staticmethod  
    def TotalInstances():  
        return T.__counter  
  
>>> T.TotalInstances()  
>>> x = T()  
>>> x.TotalInstances()
```



Statische Methoden

```
class RationalNumber:  
    #class field, will be shared by all the instances  
    numberofInstances = 0  
  
    def __init__(self,n,m):  
        """  
            Initialize the rational number  
            n,m - integer numbers  
        """  
        self.n = n  
        self.m = m  
        RationalNumber.numberofInstances+=1  
  
    @staticmethod  
    def getTotalNumberOfInstances():  
        """  
            Get the number of instances created in the app  
        """  
        return RationalNumber.numberofInstances  
  
    def testNumberOfInstances():  
        """  
            test function for getTotalNumberOfInstances  
        """  
        assert RationalNumber.getTotalNumberOfInstances() == 0  
        r1 = RationalNumber(2, 3)  
        assert RationalNumber.getTotalNumberOfInstances() == 1  
  
    testNumberOfInstances()
```



Hooks

- bisher wurden einige Operatoren vorgestellt: +, -, ...
- in Python gibt es aber gar keine Operatoren, sondern nur Operationen:
 - der „*“-Operator ruft beispielsweise intern die `__mul__`-Methode des ersten Operanden auf
 - diese speziellen Methoden kann man selbst definieren, um damit die Funktionalität zu ändern oder zu erweitern



Hooks

```
class Rational:  
    def __init__(self, num, den):  
        self.num = num  
        self.den = den  
  
    def __mul__(self, other):  
        num = self.num * other.num  
        den = self.den * other.den  
        return Rational(num, den)  
  
    def __repr__(self):  
        return "R(" + str(self.num) + "," + str(self.den) + ")"  
  
    def __str__(self):  
        return str(self.num) + "/" + str(self.den)
```



```
>>> r1 = Rational(1,2)  
>>> r2 = Rational(3,4)  
>>> r1 * r2  
R(3, 8)  
>>> print(r1 * r2)  
3/8
```



Hooks

- Vergleichsoperatoren (Rückgabe: True / False):
 - `__eq__` → `==`
 - `__ge__` → `>=`
 - `__gt__` → `>`
 - `__le__` → `<=`
 - `__lt__` → `<`
 - `__ne__` → `!=`
- `__bool__`: gilt das Objekt als Wahr oder Falsch? (Gibt True oder False zurück)



Hooks

- Numerische Operationen:
 - `__add__` → +
 - `__div__` → /
 - `__mul__` → *
 - `__sub__` → -
 - `__mod__` → %
- Element-Zugriff für Sammeltypen:
 - `__getitem__(self, index)` → `x[i]`
 - `__setitem__(self, index, value)` -> `x[i] = 10`



Hooks

```
class Data:  
    def __init__(self):  
        self.data = []  
  
    def add_element(self, elem):  
        self.data.append(elem)  
  
    def print_elements(self):  
        for el in self.data:  
            print(el)  
  
    def __getitem__(self, index):  
        return self.data[index]  
  
    def __setitem__(self, index, value):  
        self.data[index] = value  
  
data = Data()  
  
for i in range(10):  
    data.add_element(i)  
  
print ('second element', data[1])  
data[1] = 101  
data.print_elements()
```

```
second element: 1  
0  
101  
2  
3  
4  
5  
6  
7  
8  
9
```



Beispiel

```

class R:
    def __init__(self, a, b):
        if b == 0:
            raise ZeroDivisionError("b cannot be 0")
        if isinstance(a, int) and isinstance(b, int):
            self._a = a
            self._b = b
        else:
            raise ValueError("args should be int")

    def __add__(self, o):
        return T(self.a*o.b+self.b*o.a, self.b*o.b)

    def __lt__(self, o):
        return self.a/self.b < o.a/o.b

    def __eq__(self, o):
        return self.a == o.a and self.b == o.b

```

```

def __str__(self):
    if self.b == 1:
        return str(self._numerator)
    return "%i/%i" % (self.a, self.b)

@property
def a(self):
    return self._a

@property
def b(self):
    return self._b

```



Beispiel

```
def main():
    r = R(1,2)

    print(R(1,2) == R(1,2))

    print (r + R(1,2))

    print (r < R(2,3))

    try:
        p = R(1,0)
    except ZeroDivisionError:
        print('Fraction not Valid')
    except ValueError:
        print('Types not Valid')

    print('here')
main()
```

```
11     * as serviceWorker from './serviceWorker'
12
13
14
15
16
17
18
19
20 ReactDOM.render(
  <BrowserRouter>
    <Switch>
      <Route path="/login" component={Login} />
      <ProtectedRoute exact={true} path="/" component={Dashboard} />
      <ProtectedRoute path="/settings" component={Settings} />
      <ProtectedRoute component={Dashboard} />
    </Switch>
  </BrowserRouter>
), document.getElementById('root'));
```



Codestruktur

- was ist das genau?
- man benutzt einige Design Prinzipien, um Code besser strukturieren zu können
- Single Responsibility Prinzip
- Separation of Concerns
- Dependencies
- Coupling and Cohesion



Single-Responsibility-Prinzip

- Jede Funktion sollte für eine Sache verantwortlich sein
- Jede Klasse sollte eine Entität darstellen
- Jedes Modul sollte einem Aspekt der Anwendung entsprechen





Single-Responsibility-Prinzip

- lass uns das folgende Beispiel nehmen

```
def filterScore(scoreList):  
    st = input("Start score :")  
    end = input("End score:")  
    for score in scoreList :  
        if score [1] > st and score [1] < end:  
            print(score)
```



Single-Responsibility-Prinzip

- lass uns das folgende Beispiel nehmen

```
def filterScore(scoreList):  
    st = input("Start score :")  
    end = input("End score:")  
  
    for score in scoreList :  
        if score [1] > st and score [1] < end:  
            print(score)
```

- liest etwas von der Tastatur ein
- berechnet was
- gibt das Ergebnis aus



Single-Responsibility-Prinzip

- kann diese filterScore Funktion sich verändern?
- ein komplett anderes Format für Input
 - Konsoleanwendung (Menu)
 - GUI
 - Webseite
- ein neuer Filter
- ein anderes Fomat für Output
- → die Methode hat 3 Verantwortungen



Single-Responsibility-Prinzip

- der gleiche gilt für Module
- Module stellen Methoden zusammen, die thematisch miteinander passen
- mehrere Verantwortungen sind schwer zu
 - verstehen
 - verwenden
 - testen
 - warten
 - weiterentwickeln



Separation of Concerns

- man soll das Programm in verschiedenen Abschnitte aufteilen
- jeder Abschnitt adressiert ein bestimmtes Problem
- Concerns - Informationen, die auf Code auswirken
 - z. B. Computerhardware, auf der das Programm ausgeführt wird, Anforderungen, Funktionen und Modulnamen
- richtig implementiert führt zu einem Programm, das einfach zu testen ist und einfach wiederverwendet werden können



Separation of Concerns

- die gleiche Methode

```
def filterScore(scoreList):
    st = input("Start score :")
    end = input("End score:")
    for score in scoreList :
        if score [1] > st and score [1] < end:
            print(score)
```



Separation of Concerns - UI

- nur UI Funktionalität
- der Rest sind an filterScore delegiert

```
def filterScoreUI(scoreList):
    st = input("Start score :")
    end = input("End score:")

    result = filterScore(scoreList, st, end)

    for score in result :
        print(score)
```



Separation of Concerns - der Rest

- die Methode hat nur eine Verantwortung

```
def filterScore(scoreList, st, end):
    rez = []

    for p in lst:
        if p[1] > st and p[1] < end:
            rez.append(p)

    return rez
```



Separation of Concerns - das Testen

- die filterScore() Funktion kann so getestet werden

```
def filterScoreTest():
    lst = [["Anna", 100]]

    assert filterScore(lst, 10, 30) == []
    assert filterScore(lst, 1, 300) == lst

    lst == [["Anna"], 100], ["Ion"], 40], ["P"], 60]
    assert filterScore(lst, 3, 50) == [{"Ion"], 40}]
```



Dependency/Abhängigkeit

- was ist eine Abhängigkeit?
- **Funktionen** - Eine Funktion ruft eine andere Funktion auf
- **Klassen** - Eine Klassenmethode ruft eine Methode einer anderen Klasse auf
- **Module** - Eine Funktion eines Moduls ruft eine Funktion eines anderen Moduls auf
- Gegeben sei die folgenden Funktionen **a**, **b**, **c** und **d**.
 - **a** ruft **b** an, **b** ruft **c** an und **c** ruft **d** an
- Was kann passieren, wenn wir die Funktion **d** ändern?



Kohäsion

- wie gut eine Programmeinheit (eine Funktion/ein Modul) eine logische Aufgabe oder Einheit abbildet
- **starke Kohäsion:** alle Teile eines Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein.
- **schwache Kohäsion:** Teile eines Moduls haben keinen Bezug zu anderen Teilen
- viele Teile -> schwer zu verstehen!



Kopplung

ein Maß, das die Stärke die Verknüpfung von verschiedenen Systemen, Anwendungen, oder Softwaremodulen beschreibt

Formen von Kopplung. Am Beispiel von der Klasse X zur Klasse Y:

- X ist direkte oder indirekte Unterklasse von Y
- X hat Attribut bzw. Referenz von Typ Y
- X hat Methode, die Y referenziert (Abhängigkeit)



Generalisierung – Vererbung – Die Ist-Beziehung

- beschreibt eine Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer speziellen Klasse
- die spezialisierte Klasse ist vollständig konsistent mit der Basisklasse, enthält aber zusätzliche Informationen (Attribute, Methoden, Assoziationen)
- ein Objekt der spezialisierten Klasse kann überall dort verwendet werden, wo ein Objekt der Basisklasse erlaubt ist





Vererbung

- In objektorientierten Sprachen kann man (normalerweise) Klassen von anderen Klassen ableiten
- Die abgeleitete Klasse erbt Variablen und Methoden von der Basisklasse
 - Somit unterstützen die abgeleiteten Klassen die gleichen Methoden/Variablen wie die Basisklassen
 - und können überall dort benutzt werden, wo die Basisklasse benutzt werden kann
- So lange die abgeleiteten Klassen Methoden nicht überschreiben, verhalten sie sich in der abgeleiteten Klasse genauso wie in der Basisklasse



Vererbung

```
class Face:  
    def smile(self):  
        print(":-)")  
    def kiss(self):  
        print(":-*")  
  
class BabyFace(Face):  
    def pokeToungue(self):  
        print(":-P")  
  
>>> boy = BabyFace()  
>>> boy.pokeToungue()  
:-P  
>>> boy.kiss()  
:-*  
>>> boy.smile()  
:-)
```



Vererbung

```
class Person:  
def __init__(self, name):  
    self.name = name  
  
class KlingonGuy(Person):  
    def sayHello(self):  
        print("nuqneH "+  
              self.name)  
  
class GermanGuy(Person):  
    def sayHello(self):  
        print("Hallo " +  
              self.name)
```

```
>>> g =  
GermanGuy('Stefan')  
>>> g.sayHello()  
Hallo Stefan  
>>> k =  
KlingonGuy("D'utoz")  
>>> k.sayHello()  
nuqneH D'utoz
```



Clean Code

Intro

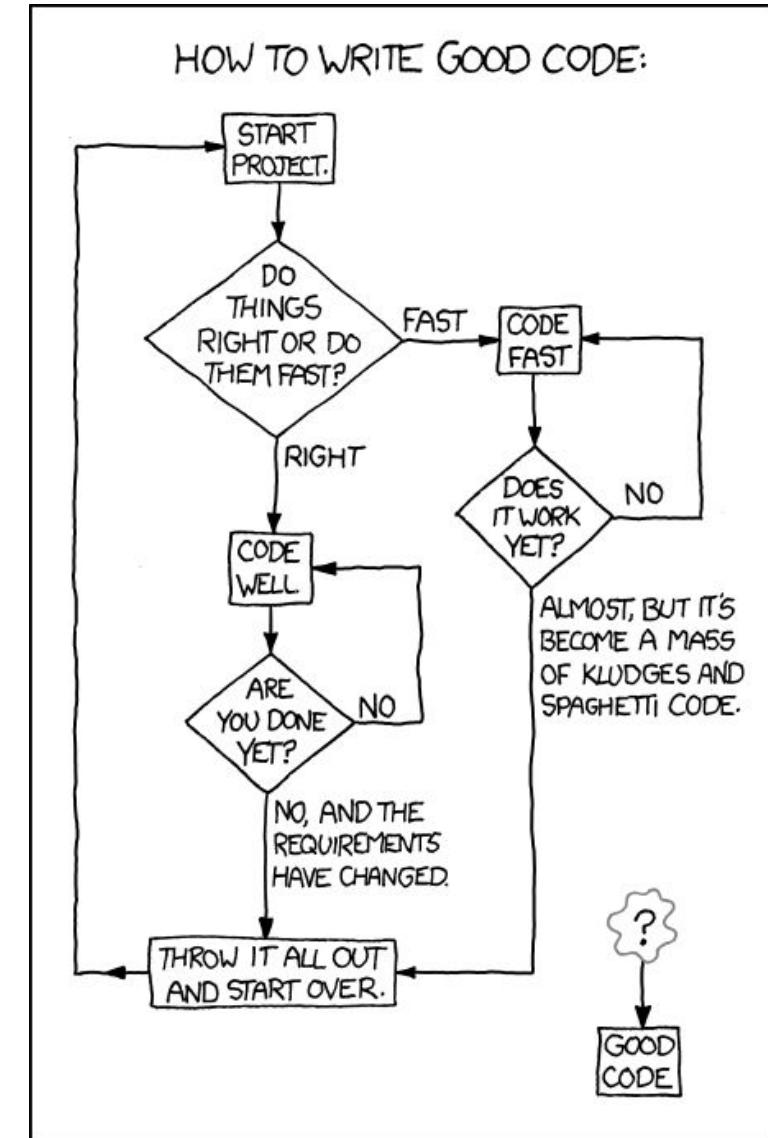






Inhalt

- Schichtenarchitektur
- GUI
- Assoziationen vs Vererbung
- Testing
- Refactoring
- Clean Code





Dynamische Typisierung

- Typprüfungen werden hauptsächlich zur Laufzeit eines Programms durchgeführt

Duck-Typing

- der Typ eines Objektes wird nicht durch seine Klasse beschrieben wird
- sondern durch vorhandene Methoden oder Attribute



Duck-Typing

- 'Wenn ich einen Vogel sehe,
 - der wie eine Ente läuft,
 - wie eine Ente schwimmt
 - und wie eine Ente schnattert,
 - dann nenne ich diesen Vogel eine Ente.'





Duck-Typing

```
class Bird:  
    def __init__(self, name):  
        self.name = name  
  
    def __str__(self):  
        return self.__class__.__name__ + ' ' + self.name  
  
class Duck(Bird):  
    def quak(self):  
        print(str(self) + ': quak')  
  
def main():  
    ducks = [Bird('Bob'), Duck('Donald'), object()]  
    for duck in ducks:  
        try:  
            duck.quak()  
        except AttributeError:  
            print('Keine Ente', duck)
```



Dateien und Python

- benötigt man die `open()`-Funktion
- Mit der `open`-Funktion erzeugt man ein Dateiobjekt
 - und liefert eine Referenz auf dieses Objekt als Ergebniswert zurück

`open(filename, mode)`

Mode: "r" , "w", "a"



Dateien und Python

Methoden

- `write(str)`
- `readline()`
- `readlines()`
- `read()`
- `close()`

Exception

- `IOError`

```
#open file for write (overwrite if exists, create if not)
f = open("test.txt", "w")
f.write("Test data\n")
f.close()
```

```
#open file for write (append if exist, create if not)
f = open("test.txt", "a")
f.write("Test data line 2\n")
f.close()
```

```
#open for read
f = open("test.txt", "r")
#read a line from the file
line = f.readline()
print line
f.close()
```

```
#open for read
f = open("test.txt", "r")
#read a line from the file
line = f.readline().strip()
while line!="":
    print line
    line = f.readline().strip()
f.close()
```

```
#open for read
f = open("test.txt", "r")
#read the entire content from the file
line = f.read()
print line
f.close()
```



Pickle

```

studenten = [('Peter', 123), ('Paula', 988), ('Freddi', 851)]

datei = open("studenten.dat", "w")
for student in studenten:
    s = str(student) + '\n'
    datei.write(s)
datei.close()

datei = open("studenten.dat", "r")
for z in datei:
    name, matnum = z.strip('()\n').split(',')
    name = name.strip("\' ")
    matnum = matnum.strip()
    print "Name: %s Matrikelnummer: %s" % (name, matnum)
datei.close()

```

konvertiert Objekte in einen Stream,
damit sie gespeichert und erneut
gelesen werden können

```

import pickle

studenten = [('Peter', 123), ('Paula', 988), ('Freddi', 851)]

datei = open("studenten.dat", "w")
pickle.dump(studenten, datei)
datei.close()

datei = open("studenten.dat", "r")
meine_studenten = pickle.load(datei)
datei.close()

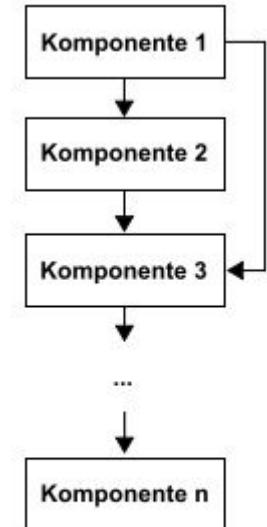
print meine_studenten

```



Schichtenarchitektur

- ein häufig angewandtes Strukturierungsprinzip für die Architektur von Softwaresystemen
- Aspekte einer „höheren“ Schicht nur solche „tieferer“ Schichten verwenden dürfen
- die Trennung von Fachkonzept, Benutzeroberfläche und Datenhaltung



Aufrufe in einer
Schichtenarchitektur



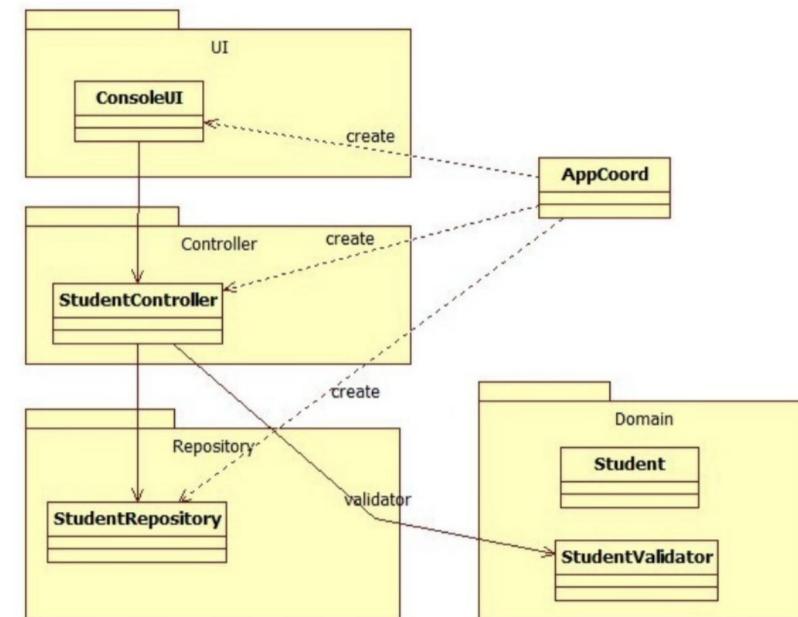
Schichtenarchitektur

- **Präsentationsschicht**
 - Benutzerschnittstelle
- **Businessschicht**
 - Controller
 - Entities
- **Datenhaltungsschicht**
 - Daten Laden und Speichern
 - Repositories



PSA: GUIs

- Tk toolkit
- Tkinter: die Python-Schnittstelle oder Interface zu Tk
- https://python-textbook.readthedocs.io/en/1.0/Introduction_to_GUI_Programming.html
- 2020.V8.zip (code)





Zwischenprüfung - 16.12

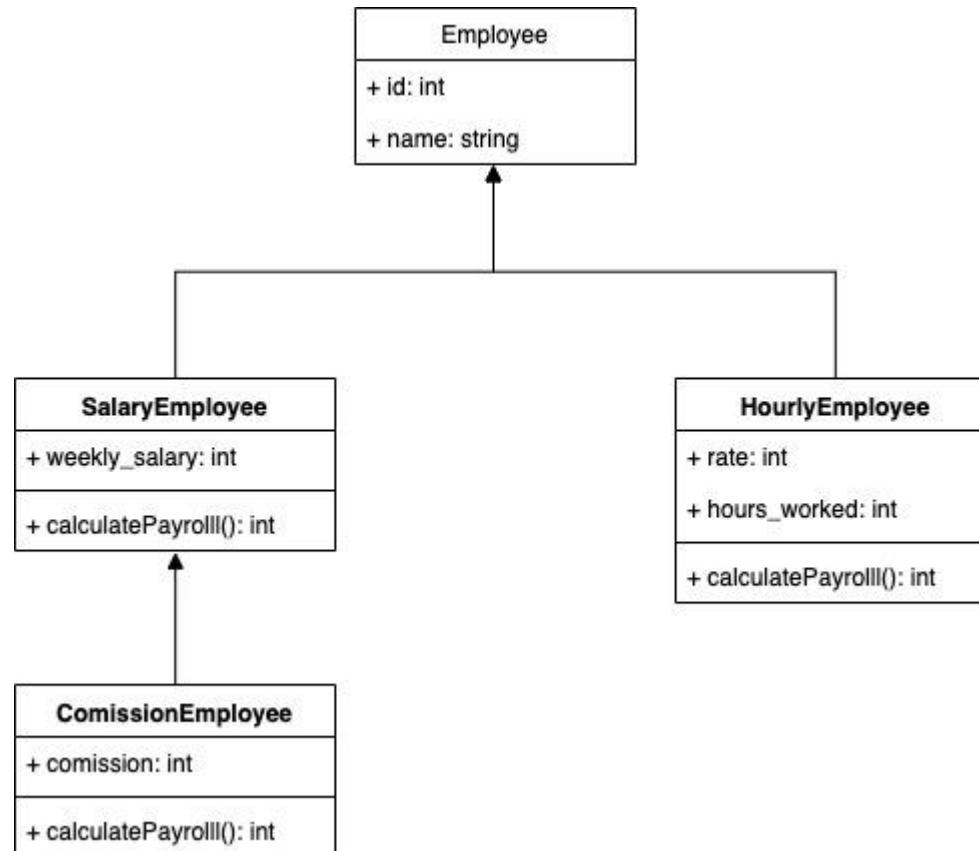
SEMINAR: $2+2=4$



KLAUSUR: DANIEL HAT EINEN APFEL. BERECHNE DIE MASSE DER SONNE



Vererbung - ist-a Beziehung





Vererbung - ist-a Beziehung

```

class Employee:
    def __init__(self, id, name):
        self.id = id
        self.name = name

class SalaryEmployee(Employee):
    def __init__(self, id, name, weekly_salary):
        super().__init__(id, name)
        self.weekly_salary = weekly_salary

    def calculate_payroll(self):
        return self.weekly_salary

class HourlyEmployee(Employee):
    def __init__(self, id, name, hours_worked, hour_rate):
        super().__init__(id, name)
        self.hours_worked = hours_worked
        self.hour_rate = hour_rate

    def calculate_payroll(self):
        return self.hours_worked * self.hour_rate

class CommissionEmployee(SalaryEmployee):
    def __init__(self, id, name, weekly_salary, commission):
        super().__init__(id, name, weekly_salary)
        self.commission = commission

    def calculate_payroll(self):
        fixed = super().calculate_payroll()
        return fixed + self.commission

```



Vererbung - ist-a Beziehung

```
class PayrollSystem:

    def calculate_payroll(self, employees):
        print('Calculating Payroll')
        print('=====')
        for employee in employees:
            print(f'Payroll for: {employee.id} - {employee.name}')
            print(f'- Check amount: {employee.calculate_payroll()}')
            print('')

salary_employee = hr.SalaryEmployee(1, 'John Smith', 1500)
hourly_employee = hr.HourlyEmployee(2, 'Jane Doe', 40, 15)
commission_employee = hr.CommissionEmployee(3, 'Kevin Bacon', 1000, 250)

payroll_system = PayrollSystem()
payroll_system.calculate_payroll([
    salary_employee,
    hourly_employee,
    commission_employee
])
```

Calculating Payroll
=====

Payroll for: 1 - John Smith
- Check amount: 1500

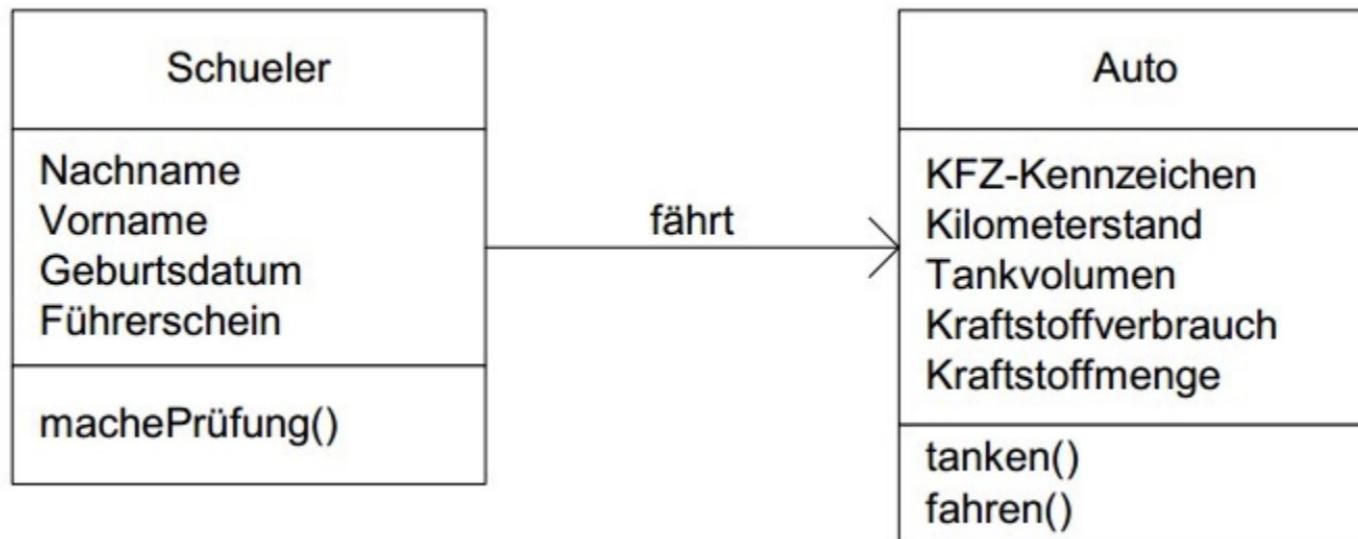
Payroll for: 2 - Jane Doe
- Check amount: 600

Payroll for: 3 - Kevin Bacon
- Check amount: 1250



Assoziation – kennt-Beziehung

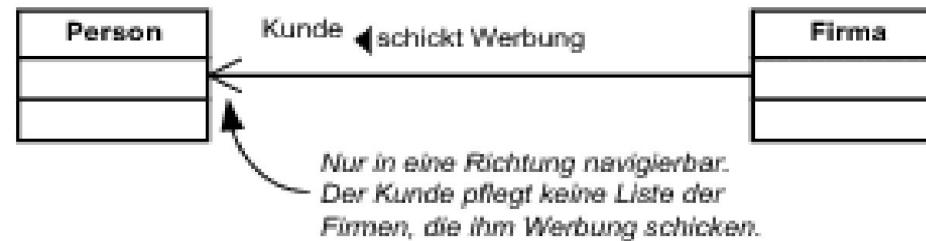
- Zwischen Objekten von Klassen können konkrete Beziehungen bestehen
- Name
- Navigierbarkeit
- Multiplizität





Navigierbarkeit

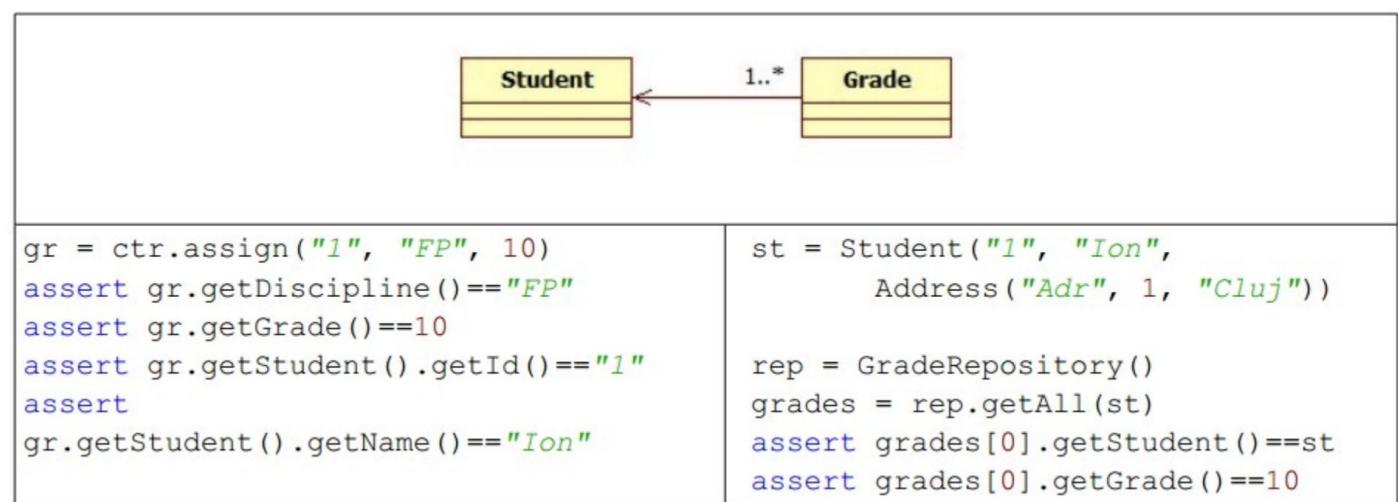
- die **Firma** kann alle **Personen** auflisten, denen sie Werbung zuschickt
- eine **Person** hat keine Liste der **Firmen**
- die **schickt Werbung** Assoziation ist navigierbar in der Richtung **Firma-Person**





Multiplizität

- wie viele Objekte des jeweiligen Typs in einer Beziehung stehen können
- Ein Objekt steht in einer Beziehung mit
 - genau einem anderen Objekt
 - keinem oder einem anderen Objekt
 - mindestens einem anderen Objekt
 - beliebig vielen anderen Objekten

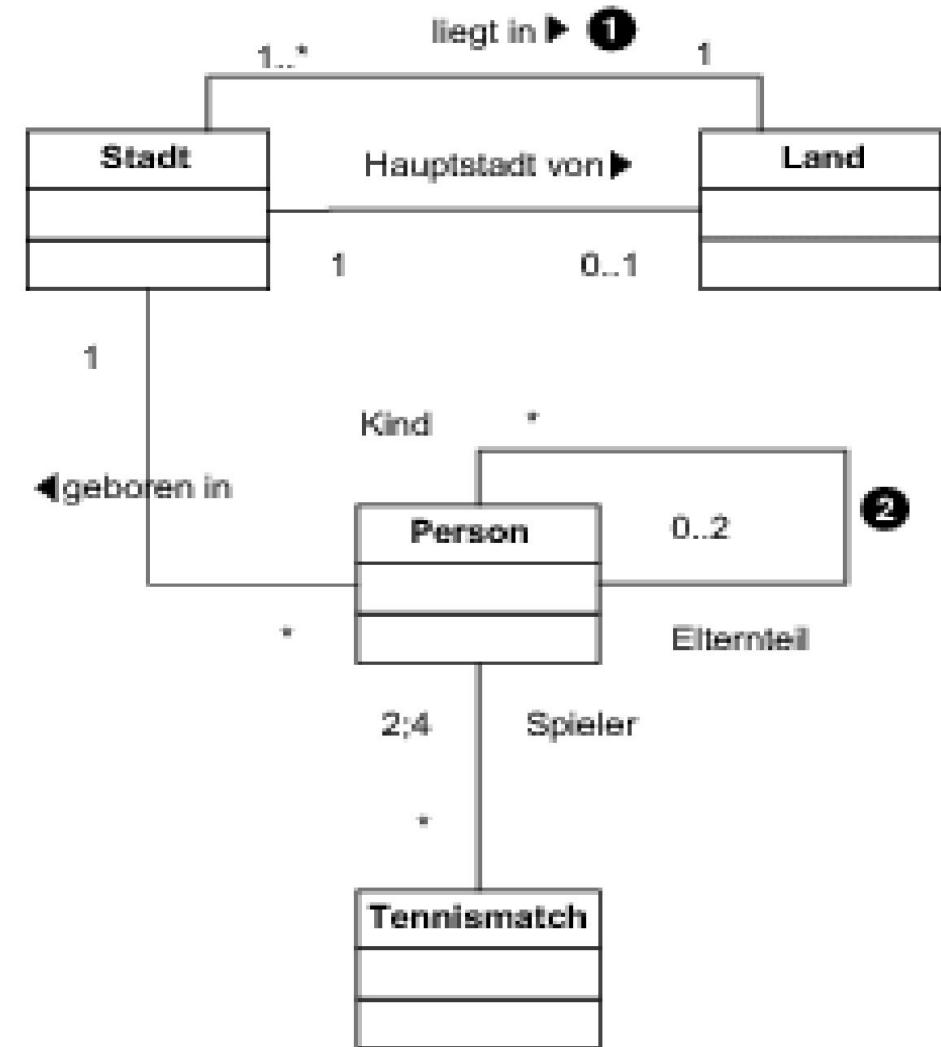




Übung

Klassen:

- Stadt
- Land
- Person
- Tennismatch





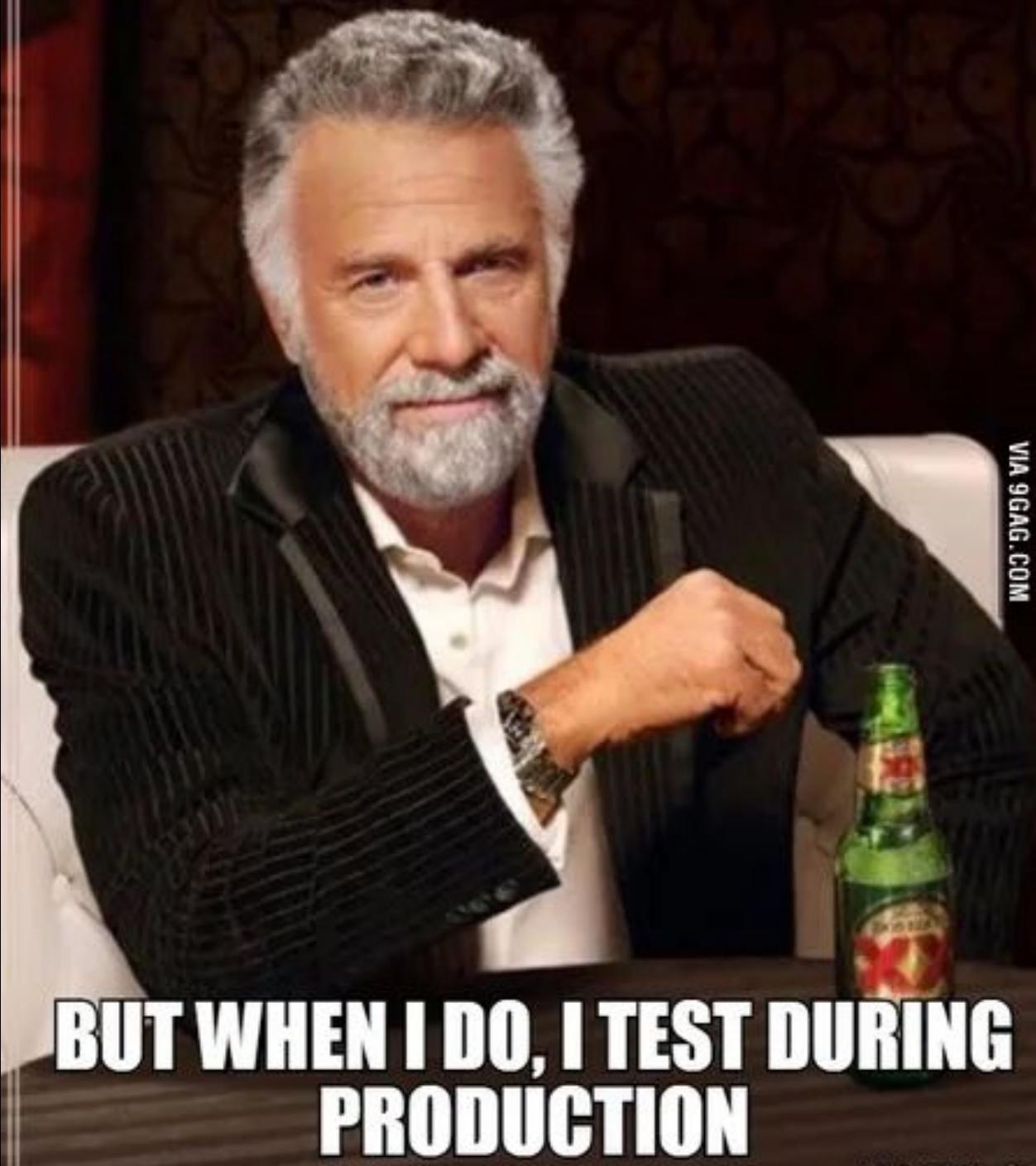


Was ist ein Fehler?

1. der Programmierer macht einen Fehler
2. und hinterlässt den Fehler im Programmcode
3. wird dieser Code ausgeführt, haben wir eine Abnormalität im Programmzustand,
4. die sich als ein Fehler nach außen manifestiert



I DON'T ALWAYS TEST



VIA 9GAG.COM

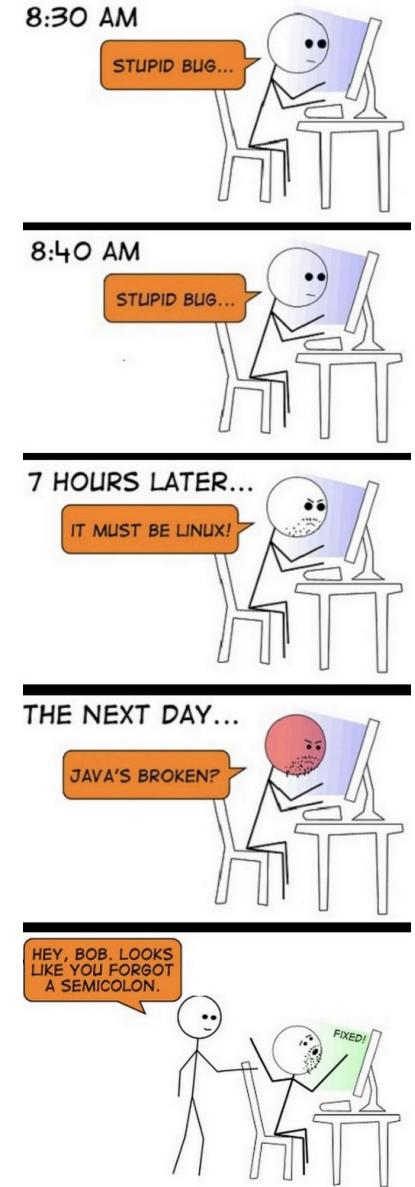
BUT WHEN I DO, I TEST DURING
PRODUCTION



Softwaretest

- Ziel des Testens ist, durch gezielte Programmausführung Fehler zu erkennen
 - Test Cases (input + output + assert)
- Das Testen soll Vertrauen in die Qualität der Software schaffen
- Die Korrektheit eines Programms kann durch Testen (außer in trivialen Fällen) nicht bewiesen werden

Program testing can be used to show the presence of bugs, but never to show their absence. (Dijkstra)





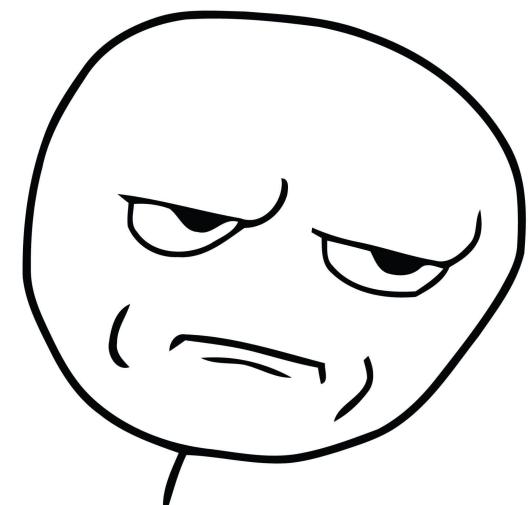
Methoden beim Testen

Black-Box-Test

die Tests **ohne Kenntnisse** über die innere Funktionsweise des zu testenden Systems entwickelt werden

White-Box-Test

die Tests **mit Kenntnissen** über die innere Funktionsweise des zu testenden Systems entwickelt werden





Methoden beim Testen

Black-Box-Test	White-Box-Test
<ul style="list-style-type: none">• Testfälle gehen von der Spezifikation aus• Interna des Testobjekts sind bei der Ermittlung der Testfälle unbekannt• Testüberdeckung wird an Hand des spezifizierten Ein/Ausgabeverhaltens gemessen	<ul style="list-style-type: none">• Testfälle ausgehend von der Struktur des Testobjekt• Testfälle werden vom Entwickler beschrieben• Testüberdeckung wird an Hand des Codes gemessen

```

def isPrime(nr):
    """
        Verify if a number is prime
        return True if nr is prime False if not
        raise ValueError if nr<=0
    """
    if nr<=0:
        raise ValueError("nr need to be positive")
    if nr==1:#1 is not a prime number
        return False
    if nr<=3:
        return True
    for i in range(2,nr):
        if nr%i==0:
            return False
    return True

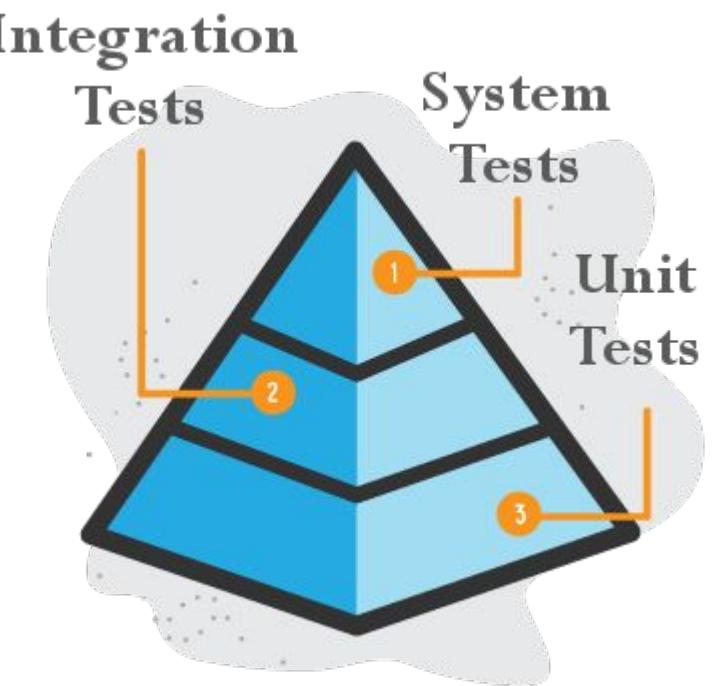
```

<p>Black Box</p> <ul style="list-style-type: none"> • test case for a prime/not prime • test case for 0 • test case for negative number 	<p>White Box (cover all the paths)</p> <ul style="list-style-type: none"> • test case for 0 • test case for negative • test case for 1 • test case 3 • test case for prime (no divider) • test case for not prime
<pre> def blackBoxPrimeTest(): assert (isPrime(5)==True) assert (isPrime(9)==False) try: isPrime(-2) assert False except ValueError: assert True try: isPrime(0) assert False except ValueError: assert True </pre>	<pre> def whiteBoxPrimeTest(): assert (isPrime(1)==False) assert (isPrime(3)==True) assert (isPrime(11)==True) assert (isPrime(9)==True) try: isPrime(-2) assert False except ValueError: assert True try: isPrime(0) assert False except ValueError: assert True </pre>



Testen

- Komponententest, Modultest (Unit Test)
 - die Tests die wir schon geschrieben haben
- Integrationstest (Integration Test)
- Systemtest (System Test)





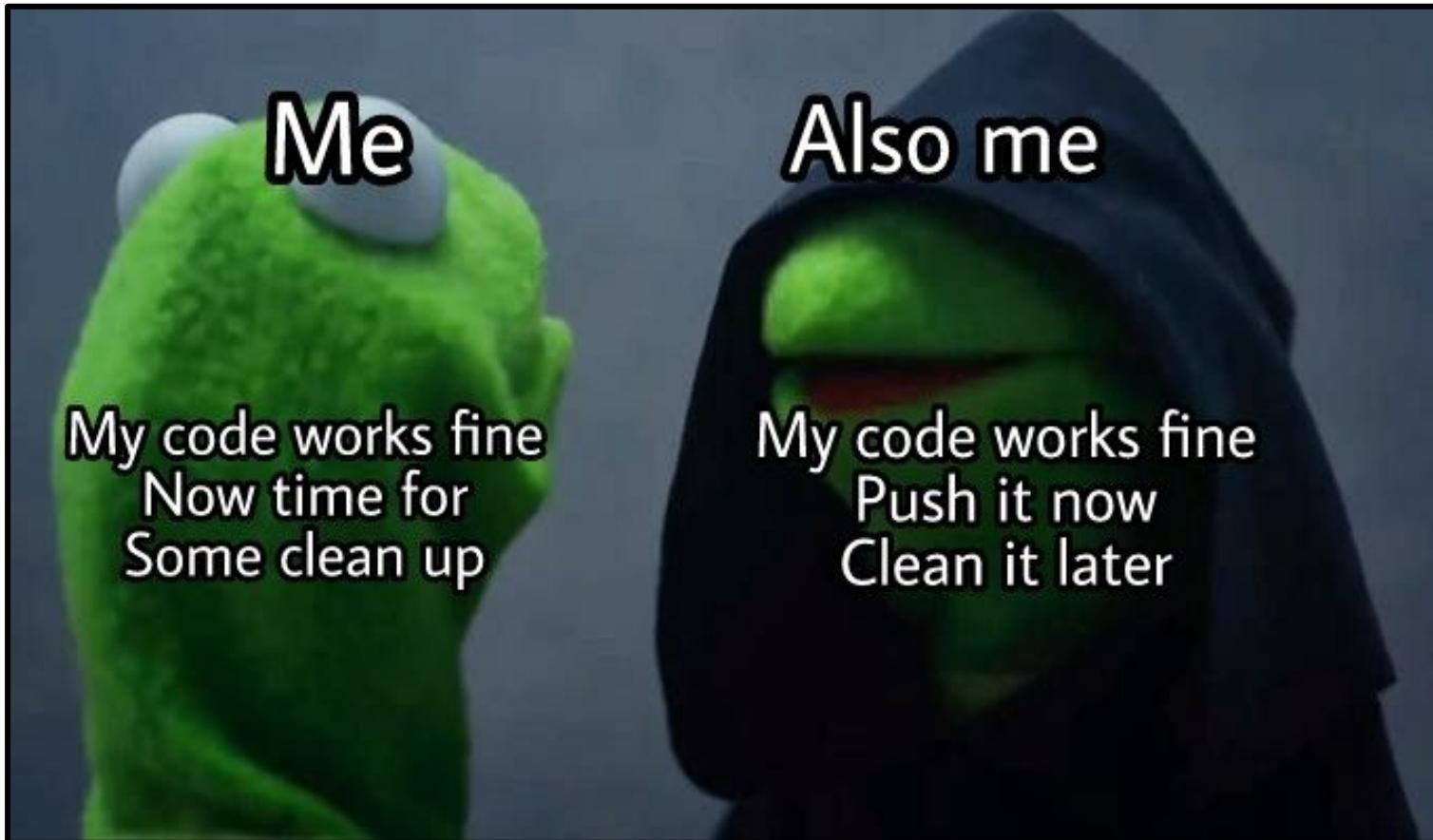
Unit Testing

- Unit Test: **Automatischer White-Box Test** welcher eine Einheit (z.B. Modul, Klasse, Komponente etc.) testet
- Unit Testing: Erstellen, Verwalten und Ausführen aller Unit Tests
- Unit Testing ist das Fundament aller agilen Softwareentwicklung Methodologien.



Clean Code

Clean Code





Refactor

$$\begin{aligned}5(x - 2) + 6(x - 2)^2 &= \\&= (x - 2)[5 + 6(x - 2)] \\&= (x - 2)(6x - 7)\end{aligned}$$

Re•fac•to•ring

(noun)

„a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.“

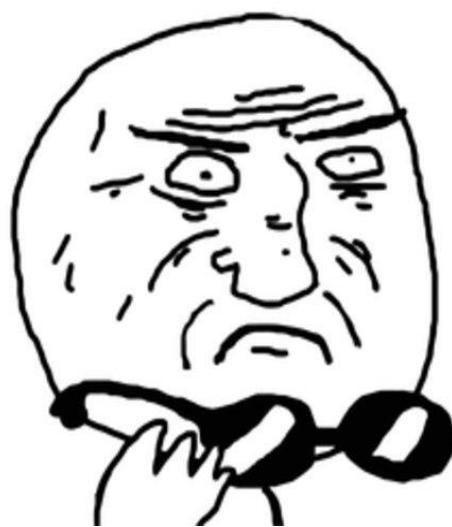
-refactoring.com



wie oft?



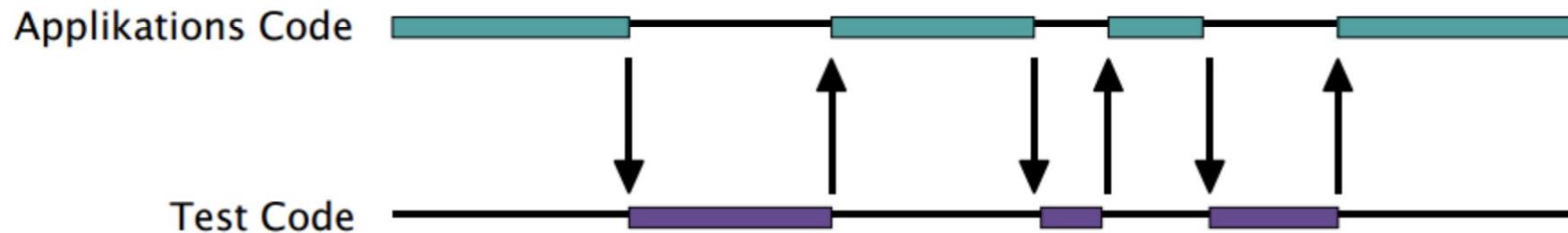
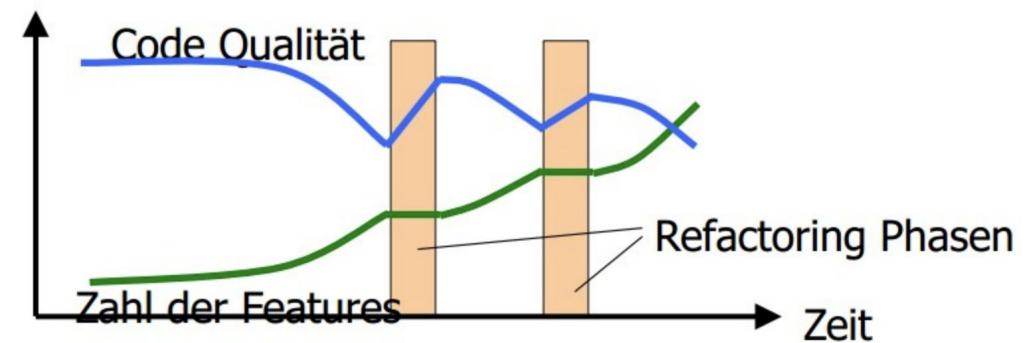
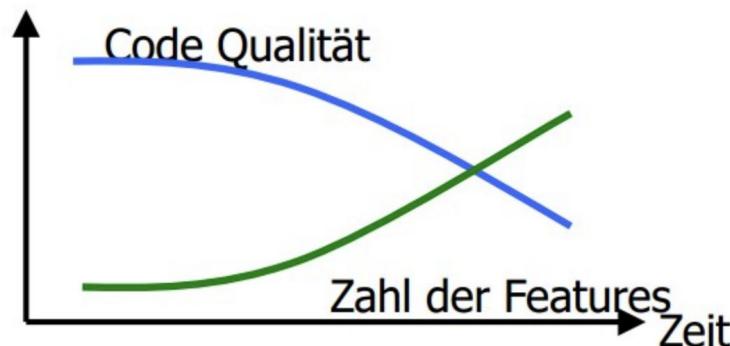
...immer



„Code a little, test a little, refactor a little!“

Refactoring:

Verbesserung des Codes ohne Änderung des Verhaltens.





4 Prinzipien

- erstelle Code, der von allen Softwareentwickler verstanden werden kann
- die Personen, die den Code reviewen/verwenden, sollten nichts annehmen
- Manchmal geht es um das Entfernen von Code (nach dem Motto: *when less is more*)
- Es gibt kein perfekter Code
 - Es gibt immer Raum für Verbesserungen



Ziele des Refactoring

- Lesbarkeit des Codes erhöhen
 - Refactoring kann parallel zu einem Code Review erfolgen
- Design verbessern (sogenannte „Bad Smells“ beseitigen)
- Code so vorbereiten, dass neue Features implementiert werden können.



not only good news...

- Refactoring ist riskant
 - Risiko minimieren durch gute Unit Test Abdeckung
- Immer in kleinen Schritten:
 - Ein Refactoring Schritt
 - Testen
 - Nächster Refactoring Schritt
 - Testen
- Häufiger Wechsel zwischen Implementation eines neuen Features und Refactoring



The return: Bad Smell

- Duplizierter Code
 - Hoher Wartungsaufwand da Änderungen überall nachgeführt werden müssen
- Lange Methode
 - Schwierig zu verstehen
 - Schlechte Wiederverwendbarkeit
 - Folge von Code Duplikationen
- Grosse Klasse
 - Oft schlechte Wiederverwendbarkeit da die Klasse viele verschiedene Dinge machte/viele Verantwortungen
 - Folge von Code Duplikationen



The return: Bad Smell

- Lange Parameter Liste
 - Schwierig zu lesen
 - Oft schlechte Wiederverwendbarkeit
 - Gefahr des Vertauschens bei Parametern des gleichen Typs
- Switch Statements bzw. if-else-if Ketten
 - Möglicherweise unflexibel für Erweiterungen
 - Gleichartige Switch Statements: Code Duplikationen



Methode extrahieren

- Ein Codefragment kann zusammengefasst werden
- Setze das Fragment in eine Methode, deren Name den Zweck bezeichnet

Motivation:

- Verbesserung der Lesbarkeit
- Codeduplikation: Verschiedene Codefragmente tun (fast) dasselbe.



Methode extrahieren

```
void foo()
{
    // berechne Kosten
    kosten = a * b + c;
    kosten -= discount;
}
```



```
void foo()
{
    berechneKosten();
}
```

```
void berechneKosten()
{
    kosten = a * b + c;
    kosten -= discount;
}
```



Methode umbenennen

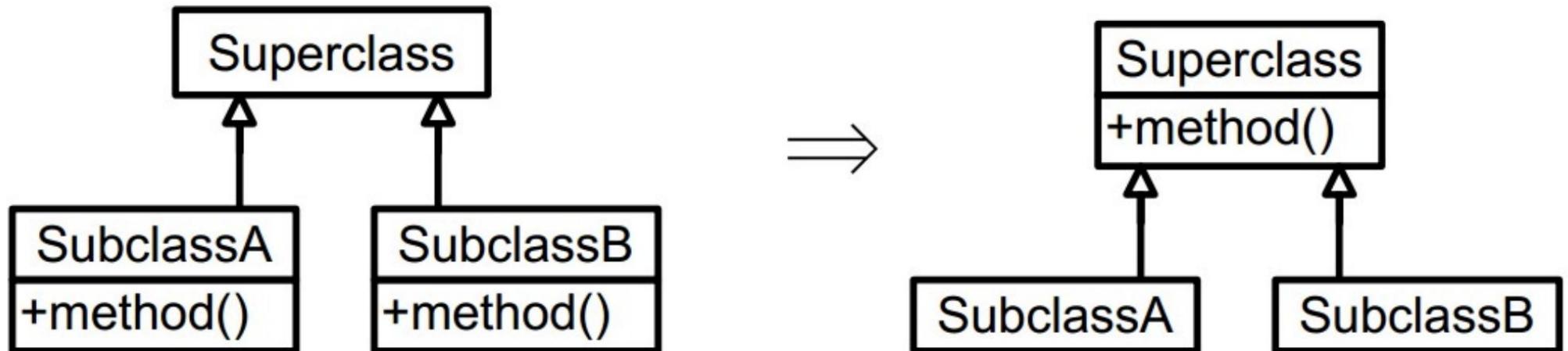
- Der Name einer Methode macht ihre Absicht nicht deutlich
- Ändere den Name der Methode





Methode hochziehen

- Es gibt Methoden mit identischen Ergebnissen in den Unterklassen
- Verschiebe die Methoden in die Oberklasse





Beschreibende Variable

- Es gibt einen komplizierten Ausdruck
- Setze den Ausdruck (oder Teile) in eine lokale Variable deren Name den Zweck erklärt.

```
if platform.toUpperCase().indexOf("MAC") > -1 and  
browser.toUpperCase().indexOf("IE") > -1 and wasInitialized() and resize > 0: #stuff
```



```
isMacOs = platform.toUpperCase().indexOf("MAC") > -1  
isIEBrowser = browser.toUpperCase().indexOf("IE") > -1  
wasResized = resize > 0;  
  
if isMacOs and isIEBrowser and wasInitialized() and wasResized: #stuff
```



Replace Temp with Query

- Eine temporäre Variable speichert das Ergebnis eines Ausdrucks
- Stelle den Ausdruck in eine Abfrage-Methode
- ersetze die temporäre Variable durch Aufrufe der Methode
- Die neue Methode kann in anderen Methoden benutzt werden.



Replace Temp with Query

```
basePrice = quantity * itemPrice;  
  
if basePrice > 1000.00: return basePrice * 0.95  
  
else: return basePrice * 0.98
```



```
if basePrice() > 1000.00:  
  
    return basePrice() * 0.95  
  
else:  
  
    return basePrice() * 0.98
```

```
def basePrice():  
  
    return quantity * itemPrice
```

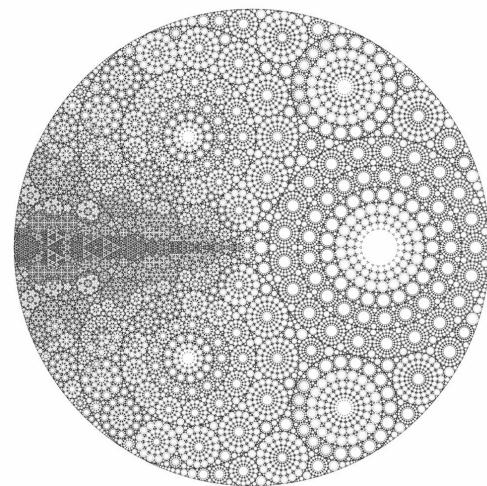


Refactoring - Methoden

- Control Flag
- Double Negative
- Zuweisung an Parametervariable



GUI + Rekursion





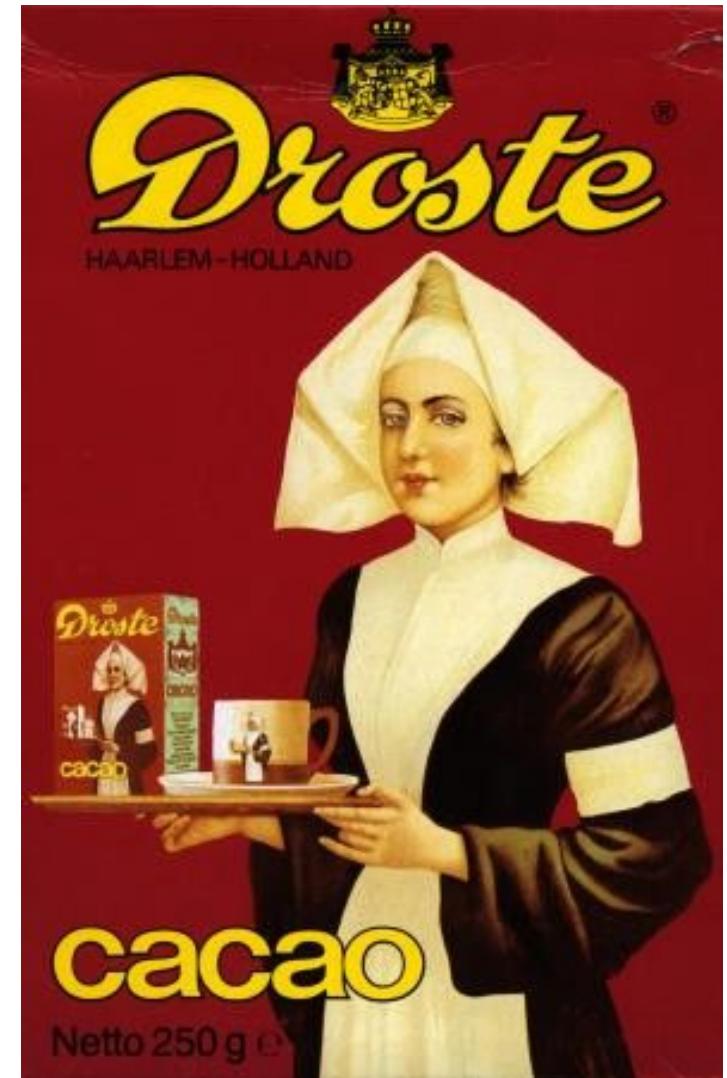
Test

- Quiz
- moodle
- <https://moodle.cs.ubbcluj.ro/>
- Grundlagen der Programmierung (FP IG)
- zoom



Inhalt

- pickle
- GUIs
- Rekursion
- Komplexität





Learning: Survival Guide

- ändere das Beispiel
- No Copy/Paste
 - Beispiele wiederholen
 - TIPPE DEN CODE EIN
- stelle (sinnvolle) Fragen
- schreibe aussagekräftige Suchanfragen





pickle





GUIs

- Tk toolkit
- Tkinter: die Python-Schnittstelle oder Interface zu Tk





GUIs für Python

- GUI: Graphical User Interface
- verschiedene GUIs verfügbar
 - **TKinter** - im Standard enthalten
 - PyGTK - basiert auf GTK
 - wxPython
 - PyQt - basiert auf Qt
 - PythonWin



- reich an Componenten
- ausreichend für mittelgroße Anwendungen
- Widgets
 - gängiges Wort für jegliche GUI-Komponente
 - frame Kontainer für alles mögliche, z.B. ein Fenster
 - canvas Leinwand/Zeichenfläche
 - button
 - checkbox
 - text
 - radiobutton
 - label z.B. Text
 - menu
 - scrollbar



Hello World

```
from tkinter import *
```

```
root = Tk()
```

```
top = Toplevel()
```

```
top.mainloop()
```



Flow

```
import tkinter as tk

class MyMainWindow:
    def __init__(self, root):
        root.title("My Window")
        root.geometry("500x500")
        self.label = tk.Label(root, text="Hello World")
        self.label.pack()
        self.button = tk.Button(root)
        self.button.configure(text="press me") # self.button ["text"] = "Press me"
        self.button.pack()

root = tk.Tk()
MyMainWindow(root)
root.mainloop()
```



Aktion Binding

- bestimmte Ereignisse erzeugen ein *event*
- z.B. Tastendruck, Mausklick, Mausbewegung, ...
- Ereignisse können an widgets (z.B. Button) gebunden werden
- Der Button wartet dann auf das jeweilige Ereignis
- tritt es auf, wird die assoziierte Funktion aufgerufen



Aktion Binding

```
import Tkinter as tk

class MyMainWindow:
    def __init__(self, root):

        root.title ("My Window")
        root.geometry("500x500")

        self.label = tk.Label (root, text = "Hello World")
        self.label.pack ()

        self.text_box = tk.Text (root, height=2, width=10)
        self.text_box.pack()

        self.add_button = tk.Button (root)
        self.add_button.configure (text = "press me")
        self.add_button.pack ()

        self.exit_button = tk.Button (root)
        self.exit_button.configure (text = "exit")
        self.exit_button.pack ()

        self.add_button.bind ("<Button -1 >", self.add_button_action)
        self.exit_button.bind ("<Button -1 >", self.exit_button_action)

    def add_button_action(self, event):
        input_text = self.text_box.get("1.0","end-1c")
        self.label.config(text=input_text)

    def exit_button_action(self, event):
        root.destroy()

root = tk.Tk ()
MyMainWindow (root)
root.mainloop ()
```



Aktion Binding

- Button reagiert nicht auf kompletten Klick
- Event <Button-1> ist gleich <ButtonPress-1>
- 1=left mouse button, 2=right, 3=middle
- loslassen entspricht <ButtonRelease-1>
- mit command werden mehrere Events an ein widget gebunden



Aktion Binding

```
import Tkinter as tk

class MyMainWindow:
    def __init__(self, root):

        root.title ("My Window")
        root.geometry("500x500")

        self.label = tk.Label (root, text = "Hello World")
        self.label.pack ()

        self.text_box = tk.Text (root, height=2, width=10)
        self.text_box.pack()

        self.add_button = tk.Button (root, text = "press me",
                                    command = self.add_button_action)
        self.add_button.pack ()

        self.exit_button = tk.Button (root, text = "exit",
                                    command = self.exit_button_action)
        self.exit_button.pack ()

    def add_button_action(self):
        input_text = self.text_box.get("1.0","end-1c")
        self.label.config(text=input_text)

    def exit_button_action(self ):
        root.destroy()

root = tk.Tk ()
MyMainWindow (root)
root.mainloop ()
```



Command

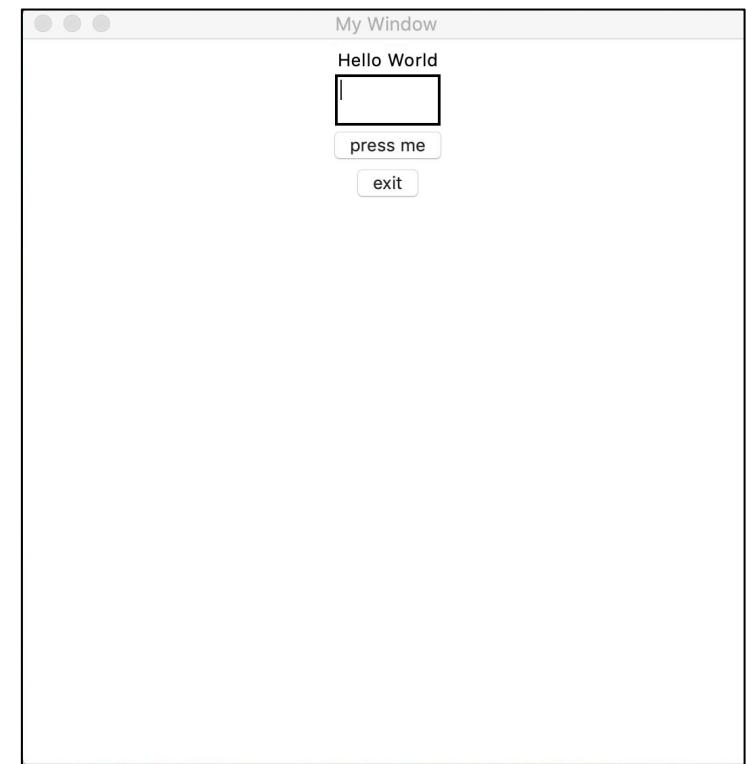
- `command=self.action1`
 - `command` ist die Funktion (deren Name)
- Mit Parameter: `command=self.action1(x)`?
 - **hier wäre command der Rückgabewert der Funktion (None)**
- Lösung: Funktion ohne Parameter angeben, die eine Funktion mit Parametern aufruft (`lambda-Funktion`)

```
command = lambda : self.action("do_delete"))
command = lambda : self.action1(var1 , var2 )
command = lambda x=var1,y=var2:self.action1(x,y)
```
- **Funktionsheader:** `def action1(self, x, y):`



widget Hierarchie

- bisher wurden alle widgets direkt als Kinder von root erzeugt
- widgets können selbst Kinder haben
- beliebige Verschachtelungen möglich
- für alle widgets kann ein Padding angegeben werden
- Geometrie-Manager:
 - pack
 - grid





Geometrie-Manager

- pack
- Anordnung im wesentlichen über
 - side=... (LEFT, TOP, ...)
- fill gibt an, wie sich die Objekte ausdehnen sollen
 - tk.NONE gar nicht
 - tk.X, tk.Y nur in X/Y-Richtung
 - tk.BOTH in beide Richtungen
- expand - widget versucht, möglichst viel Fläche zu belegen
- anchor - damit kann das widget sich in einem bestimmten Teil der Fläche platzieren
 - tk.N, tk.NW, tk.NE, ..., tk.CENTER
- grid
 - Anordnung der widgets in einem Gitter
 - z.B. `widget.grid(row=2, column=7)`



Beispiel

```
import Tkinter as tk

class MyMainWindow:
    def __init__(self, root):
        root.title ("My Window")
        root.geometry("500x500")

        self.buttons_frame = tk.Frame (root)
        self.buttons_frame.pack(side = tk.TOP, fill = tk.BOTH)

        self.add_button = tk.Button (self . buttons_frame , text = "press me",
            command = self.add_button_action)
        self.add_button.pack(side = tk.LEFT)

        self.exit_button = tk.Button (self . buttons_frame , text = "exit",
            command = self.exit_button_action)
        self.exit_button.pack (side = tk.RIGHT, fill = tk.BOTH)

        self.top = tk.Frame (root)
        self.top.pack (side = tk.TOP, fill = tk.BOTH)

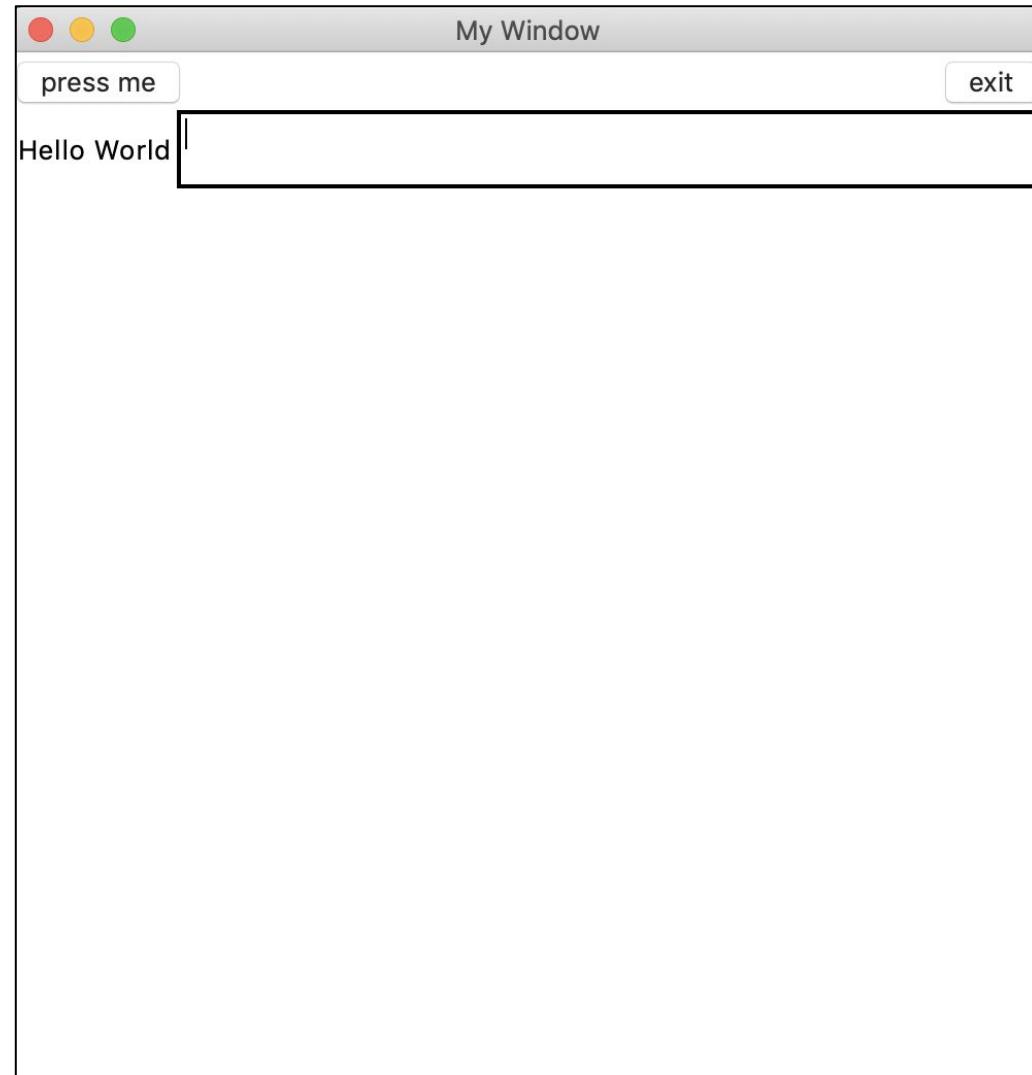
        self.label = tk.Label (self.top, text = "Hello World")
        self.label.pack (side = tk.LEFT)

        self.text_box = tk.Text (self.top, height=2, width=100)
        self.text_box.pack(side = tk.RIGHT)

    def add_button_action(self):
        input_text = self.text_box.get("1.0","end-1c")
        self.label.config(text=input_text)

    def exit_button_action(self ):
        root.destroy()

root = tk.Tk ()
MyMainWindow (root)
root.mainloop ()
```





Multi-Window





Wenn du einer der Non-Konformisten sein willst, dann musst du dich nur genau so anziehen wie wir und die gleiche Musik hören



Rekursion

- Neue Denkweise
- Wikipedia: “Als Rekursion bezeichnet man den Aufruf oder die Definition einer Funktion durch sich selbst.”
- Rekursion ist eine Form der Wiederholung
- Rekursion ermöglicht
 - elegante Algorithmen
 - Komplexitätsanalyse



Rekursion

In der Mathematik ist es oft einfacher, eine Funktion rekursiv zu definieren

$$\text{ggt}(a, b) = \begin{cases} a & \text{falls } b = 0 \\ \text{ggt}(b, a \bmod b) & \text{sonst} \end{cases}$$

- **ggt** dient zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen
- Absicht deutlich aus eigener Definition

$$\begin{aligned} \text{ggt}(33, 12) &= \text{ggt}(12, 33 \bmod 12) = \text{ggt}(12, 9) \\ &= \text{ggt}(9, 12 \bmod 9) = \text{ggt}(9, 3) \\ &= \text{ggt}(3, 9 \bmod 3) = \text{ggt}(3, 0) \\ &= 3 \end{aligned}$$



Definition

- eine Funktion nennt man **rekursiv**, wenn sie **sich selbst aufruft**
- rekursive Funktionen bestehen immer aus den folgenden
- Bestandteilen:
 - mindestens ein **Basisfall**, in dem die Rekursion abbricht und das Ergebnis entsteht
 - mindestens ein **rekursiver Fall**, in dem die Funktion sich selbst mit veränderten Argumenten aufruft

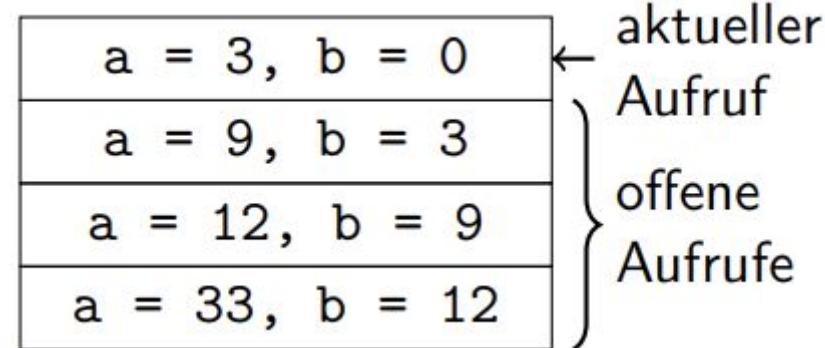
```
def ggt(a, b): //ggt = größter gemeinsamer Teiler
    if (b == 0)
        return a; //Basisfall

    return ggt(b, a % b); // rekursiver Fall
```



Rekursion und der Stack

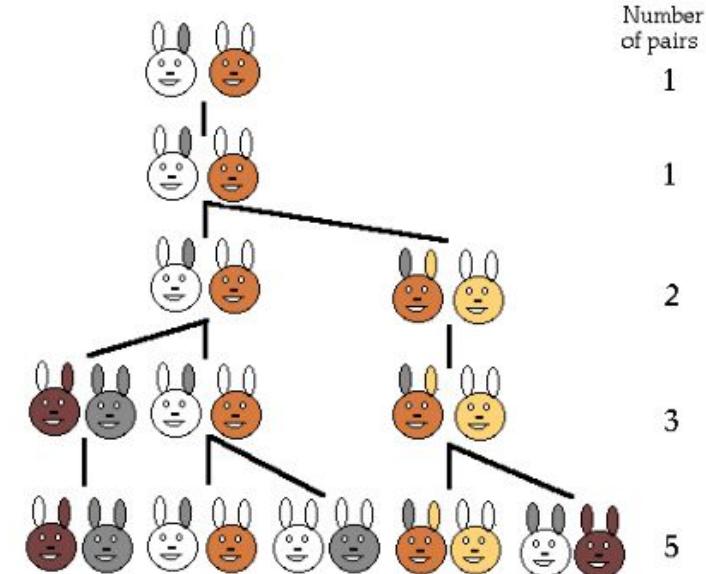
- jeder Aufruf legt einen neuen Stack Frame mit seinen Argumenten oben auf den Stack
- die aktuellen Argumentwerte stehen im obersten Frame
- bei einem return wird der Stack Frame geschlossen

$$\begin{aligned} \text{ggt}(33, 12) &= \text{ggt}(12, 9) \\ &= \text{ggt}(9, 3) \\ &= \text{ggt}(3, 0) \\ &= 3 \end{aligned}$$


Rekursion

Beginn der Fibonacci-Folge

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...



- zu Beginn eines Jahres gibt es genau ein Paar neugeborener Kaninchen
- dieses Paar wirft nach 2 Monaten ein neues Kaninchenpaar
- und dann monatlich jeweils ein weiteres Paar
- jedes neugeborene Paar vermehrt sich auf die gleiche Weise



Fibonacci-Zahlen

- Rekursive Definition der Fibonacci-Folge:
 - a. $\text{fib}(n) = 0$, falls $n = 0$
 - b. $\text{fib}(n) = 1$, falls $n = 1$
 - c. $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$, falls $n \geq 2$
- Die Rekursion in (c) stoppt, wenn $n = 0$ oder $n = 1$
- Abbruchbedingung?

```
def fibonacci():
    """
        compute the fibonacci number
        n - a positive integer
        return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```



Fakultätsfunktion

die Fakultätsfunktion:

$$4! = 4 * 3 * 2 * 1$$

rekursive Definition der Fakultätsfunktion:

- a. $f(\text{ku}(n)) = 1$, falls $n = 0$
- b. $f(\text{ku}(n)) = n * f(\text{ku}(n-1))$, falls $n > 0$

```
def factorial(n):
    """
        compute the factorial
        n is a positive integer
        return n!
    """
    if n== 0:
        return 1
    return factorial(n-1)*n
```



Rekursion

Abbruchbedingung?

Analog zu unendlichen Schleifen mit for- and while- Schleifen

```
def gogogo(x):
    print x

    if x % 2 == 0:
        return gogogo(x / 2)
    else:
        return gogogo(3 * x + 1)
```



Rekursion vs. Iteration

- rekursive Algorithmen sind oft **natürlicher** und **einfacher** zu finden
- die **Korrektheit** rekursiver Algorithmen ist oft einfacher zu prüfen
- rekursive Lösungen sind i.d.R. statisch kürzer und auch, weil verständlicher, änderung freundlicher



Rekursion vs. Iteration

jeder rekursive Algorithmus kann in einen iterativen transformiert werden

```
TailRecursiveAlgorithm (...) {  
    if condition {  
        A  
    } else {  
        B  
        TailRecursiveAlgorithm(...);  
    }  
}
```



```
IterativeAlgorithm (...) {  
    while not condition {  
        B  
    }  
    A  
}
```



Rekursion vs. Iteration

von Iteration zu Rekursion

```
IterativeAlgorithm (...) {  
    A  
    while condition {  
        B  
    }  
    C  
}
```



```
Algorithm (...) {  
    A  
    RecursiveAlgorithm(...);  
    C  
}  
  
RecursiveAlgorithm (...) {  
    if condition {  
        B  
        RecursiveAlgorithm(...);  
    }  
}
```

Übung



Schreiben Sie für ein String eine Funktion, die true zurückgibt, wenn die angegebene Zeichenkette palindrom ist, andernfalls false.





Übung

Schreiben Sie ein Programm, um einen Stapel mit Hilfe von Rekursion umzukehren.

Sie dürfen keine Schleifenkonstrukte wie while, for verwenden, und Sie können nur die folgenden Funktionen auf Stack S verwenden:

- `is_empty(S)`
- `push(S)`
- `pop(S)`





Einsatz von Rekursion

Entwurf durch Reduktion auf leichtere Probleme:

splitte ein Problem, so dass die Lösung eines bekannten einfacheren Problems auf den Rest des ursprünglichen Problems angewandt werden kann

Divide&Conquer Strategie:

teile das Problem in zwei Teilen auf und behandle jede Teile mit dem gleichen Verfahren, bis die Teile klein genug sind, um eine direkte Lösung zu erlauben

Ersetze Rekursion durch Schleifen,

wann immer dies einfach durchzuführen ist



Komplexität





Beurteilung von Algorithmen

- viele Algorithmen, um dieselbe Funktion zu realisieren
- Welche Algorithmen sind die besseren?
- nicht-funktionaler Eigenschaften:
 - Zeiteffizienz: Wie lange dauert die Ausführung?
 - Speichereffizienz: Wie viel Speicher wird zur Ausführung benötigt?
 - Benötigte Netzwerkbandbreite
 - Einfachheit des Algorithmus
 - Aufwand für die Programmierung

Ressourcenbedarf

- Prozesse verbrauchen:
 - Rechenzeit
 - Speicherplatz
- Die Ausführungszeit hängt ab von:
 - der konkreten Programmierung
 - Prozessorgeschwindigkeit
 - Programmiersprache
 - Qualität des Compilers





Beispiel

```
def fibonacci(n):
    """
        compute the fibonacci number
        n - a positive integer
        return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

```
def fibonacci2(n):
    """
        compute the fibonacci number
        n - a positive integer
        return the fibonacci number for a given n
    """
    sum1 = 1
    sum2 = 1
    rez = 0
    for i in range(2, n+1):
        rez = sum1+sum2
        sum1 = sum2
        sum2 = rez
    return rez
```

```
def measureFibo(nr):
    sw = Stopwatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    sw = Stopwatch()
    print "fibonacci(", nr, ") =", fibonacci(nr)
    print "fibonacci take " +str(sw.stop())+" seconds"
```

```
measureFibo(32)

fibonacci2( 32 ) = 3524578
fibonacci2 take 0.0 seconds
fibonacci( 32 ) = 3524578
fibonacci take 1.7610001564 seconds
```



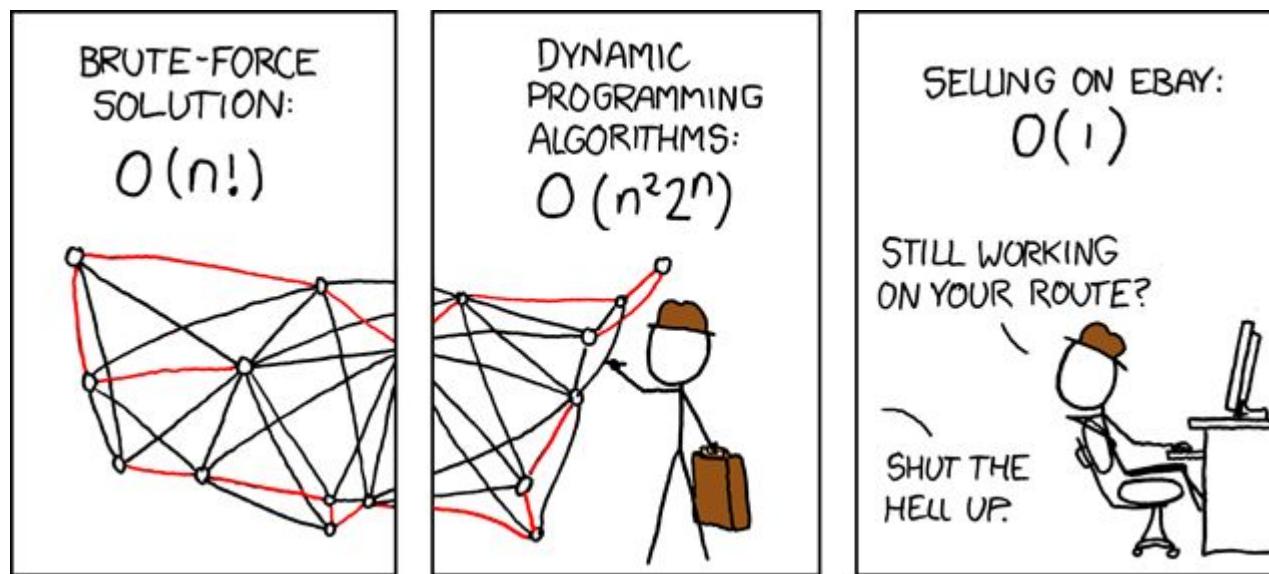
Leistungsverhalten

- **Speicherplatzkomplexität:**
 - Wird primärer & sekundärer Speicherplatz effizient genutzt?
- **Laufzeitkomplexität:**
 - Steht die Laufzeit im akzeptablen / vernünftigen / optimalen Verhältnis zur Aufgabe?
- **Theorie:**
 - liefert untere Schranke, die für jeden Algorithmus gilt, der das Problem löst
- Spezieller Algorithmus liefert obere Schranke für die Lösung des Problems

Laufzeit

Die Laufzeit $T(x)$ eines Algorithmus A bei Eingabe x ist definiert als die Anzahl von Basisoperationen, die Algorithmus A zur Berechnung der Lösung bei Eingabe x benötigt

Ziel: Laufzeit = Funktion der Größe der Eingabe





Laufzeit

- Sei P ein gegebenes Programm und x Eingabe für P , $|x|$ Länge von x , und $T(x)$ die Laufzeit von P auf x
- **Ziel:** beschreibe den Aufwand eines Algorithmus als Funktion der Größe des *Inputs*
- **Der beste Fall:**

$$T(n) = \inf \{T(x) \mid |x| = n, x \text{ Eingabe für } P\}$$

- **Der schlechteste Fall:**

$$T(n) = \sup \{T(x) \mid |x| = n, x \text{ Eingabe für } P\}$$



Minimum-Suche

Eingabe: Array von n Zahlen

Ausgabe: index i , so dass $a[i] < a[j]$, für alle j

```
def min(A):
    min = 0
    for j in range(1, len(A)):
        if A[j] < A[min]:
            min = j
    return min
```



Minimum-Suche

```
def min(A):
    min = 0
    for j in range( 1, len(A) ):
        if A[j] < A[min]:
            min = j
    return min
```

Kosten:	Max Anzahl:
c1	1
c2	n-1
c3	n-1
c4	n-1

Zeit:

$$T(n) = c_1 + (n-1)(c_2+c_3+c_4) < c_5n + c_1$$

n = Größe des Arrays



Such + Sortieralgorithmen I





Inhalt

- Komplexität
- Search
- Sort





Beurteilung von Algorithmen

- viele Algorithmen, um dieselbe Funktion zu realisieren
- Welche Algorithmen sind die besten?
- nicht-funktionaler Eigenschaften:
 - Zeiteffizienz
 - Wie lange dauert die Ausführung?
 - Speichereffizienz
 - Wie viel Speicher wird zur Ausführung benötigt?
 - Benötigte Netzwerkbandbreite
 - Einfachheit des Algorithmus
 - Aufwand für die Programmierung

Ressourcenbedarf

- Prozesse verbrauchen:
 - Rechenzeit
 - Speicherplatz
- Die Ausführungszeit hängt ab von:
 - der konkreten Programmierung
 - Prozessorgeschwindigkeit
 - Programmiersprache
 - Qualität des Compilers





Beispiel

```
def fibonacci(n):
    """
        compute the fibonacci number
        n - a positive integer
        return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

```
def fibonacci2(n):
    """
        compute the fibonacci number
        n - a positive integer
        return the fibonacci number for a given n
    """
    sum1 = 1
    sum2 = 1
    rez = 0
    for i in range(2, n+1):
        rez = sum1+sum2
        sum1 = sum2
        sum2 = rez
    return rez
```

```
def measureFibo(nr):
    sw = Stopwatch()
    print "fibonacci2(", nr, ") =", fibonacci2(nr)
    print "fibonacci2 take " +str(sw.stop())+" seconds"

    sw = Stopwatch()
    print "fibonacci(", nr, ") =", fibonacci(nr)
    print "fibonacci take " +str(sw.stop())+" seconds"
```

```
measureFibo(32)

fibonacci2( 32 ) = 3524578
fibonacci2 take 0.0 seconds
fibonacci( 32 ) = 3524578
fibonacci take 1.7610001564 seconds
```

[] ja

[] nein

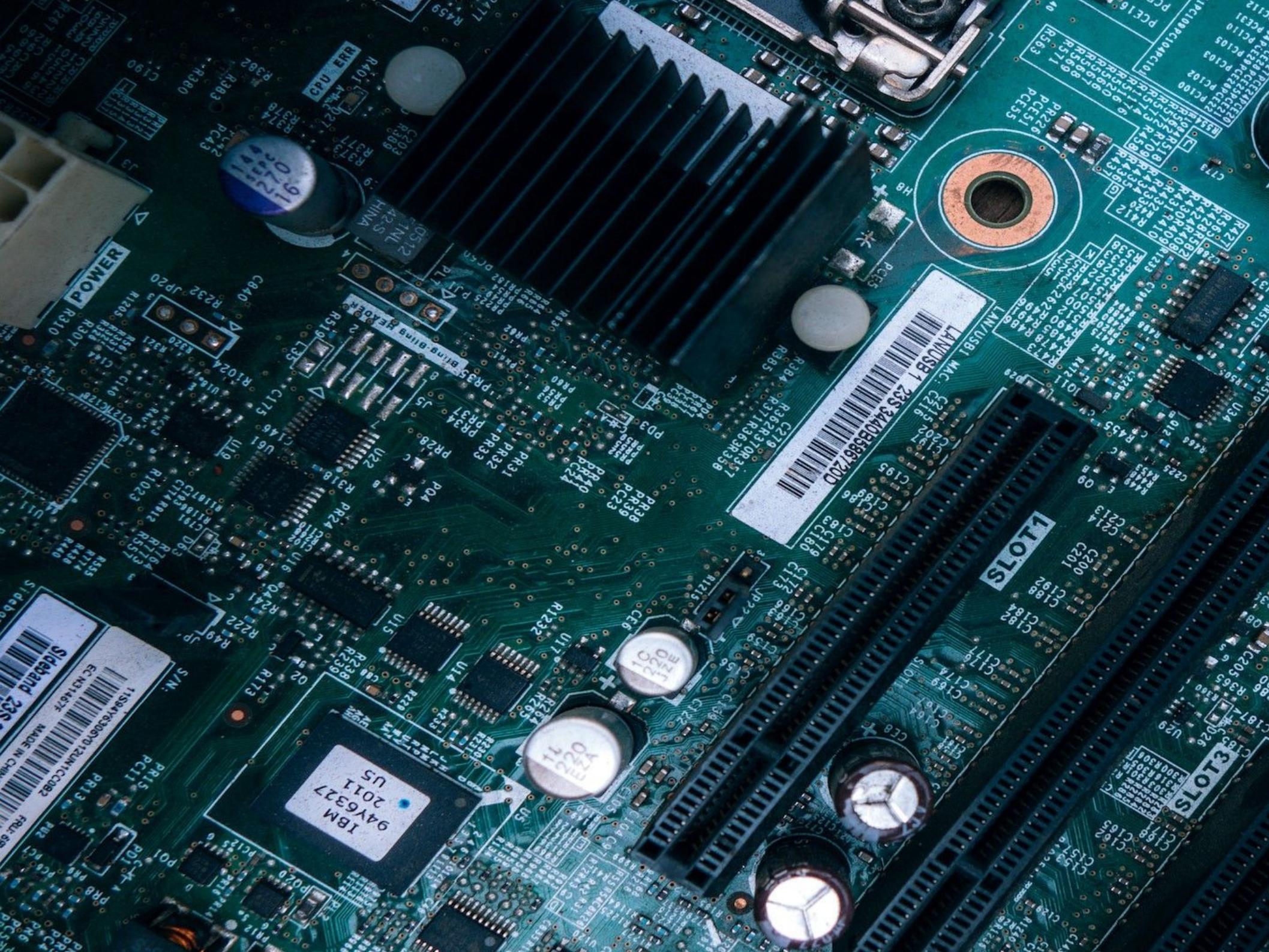
[X] ja
ein



Beispiel

```
def fibonacci(n):
    """
    compute the fibonacci number
    n - a positive integer
    return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```







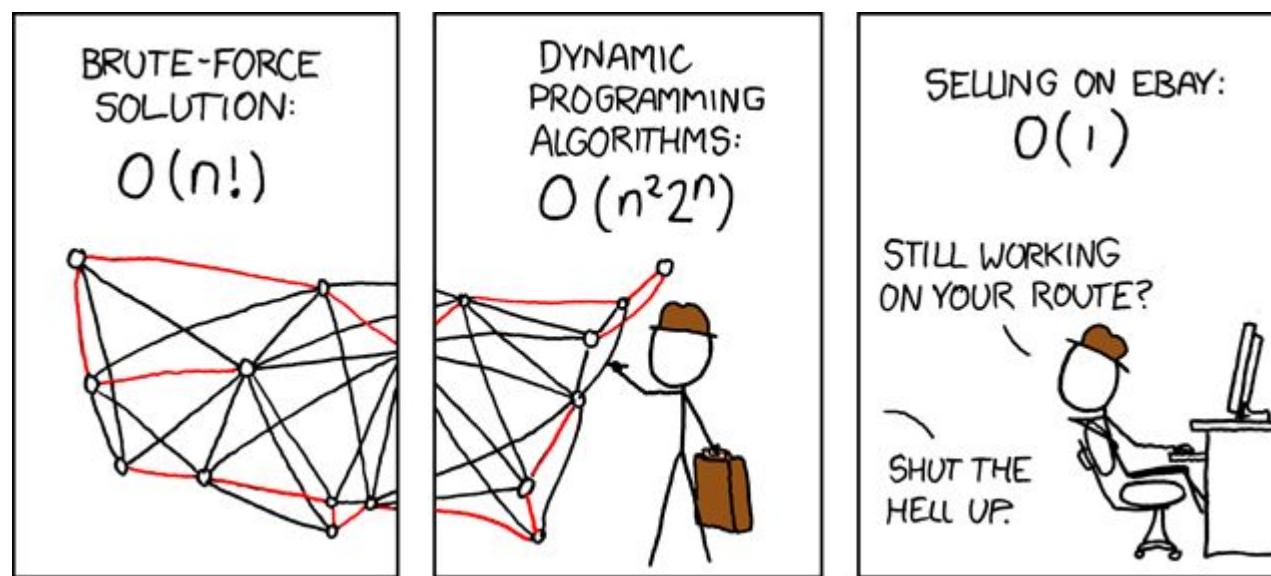
Leistungsverhalten

- **Speicherplatzkomplexität:** Wird primärer & sekundärer Speicherplatz effizient genutzt?
- **Laufzeitkomplexität:** Steht die Laufzeit im akzeptablen / vernünftigen / optimalen Verhältnis zur Aufgabe?
- **Theorie:** liefert untere Schranke, die für jeden Algorithmus gilt, der das Problem löst
- Spezieller Algorithmus liefert obere Schranke für die Lösung des Problems

Laufzeit

Die Laufzeit $T(x)$ eines Algorithmus A bei Eingabe x ist definiert als die **Anzahl von Basisoperationen**, die Algorithmus A zur Berechnung der Lösung bei Eingabe x benötigt

Ziel: Laufzeit = Funktion der Größe der Eingabe





Laufzeit

- Sei A ein gegebener Algorithmus und x Eingabe für A , $|x|$ Länge von x , und $T(x)$ die Laufzeit von A auf x
- **Ziel:** beschreibe den Aufwand eines Algorithmus als Funktion der Größe des *Inputs*
- **Der beste Fall:**

$$T(n) = \inf \{ T(x) \mid |x| = n, x \text{ Eingabe für } A \}$$

- **Der schlechteste Fall:**

$$T(n) = \sup \{ T(x) \mid |x| = n, x \text{ Eingabe für } A \}$$



Basisoperationen und deren Kosten

Für eine präzise mathematische Laufzeitanalyse benötigen wir **ein Rechenmodell**, das Basisoperationen und deren Kosten definiert.

Als Basisoperationen definieren wir:

- Arithmetische Operationen
- Datenverwaltung
- Kontrolloperationen

Kosten: Zur Vereinfachung nehmen wir an, dass jede dieser Operationen bei allen Operanden gleich viel Zeit benötigt



Minimum-Suche

Eingabe: Array von n Zahlen

Ausgabe: index i , so dass $a[i] < a[j]$, für alle j

```
def min(A):
    min = 0
    for j in range(1, len(A)):
        if A[j] < A[min]:
            min = j
    return min
```



Minimum-Suche

```
def min(A):
    min = 0
    for j in range( 1, len(A) ):
        if A[j] < A[min]:
            min = j
    return min
```

Kosten:	Max Anzahl:
c1	1
c2	n-1
c3	n-1
c4	n-1

Zeit:

$$T(n) = c_1 + (n-1)(c_2+c_3+c_4) < c_5n + c_1$$

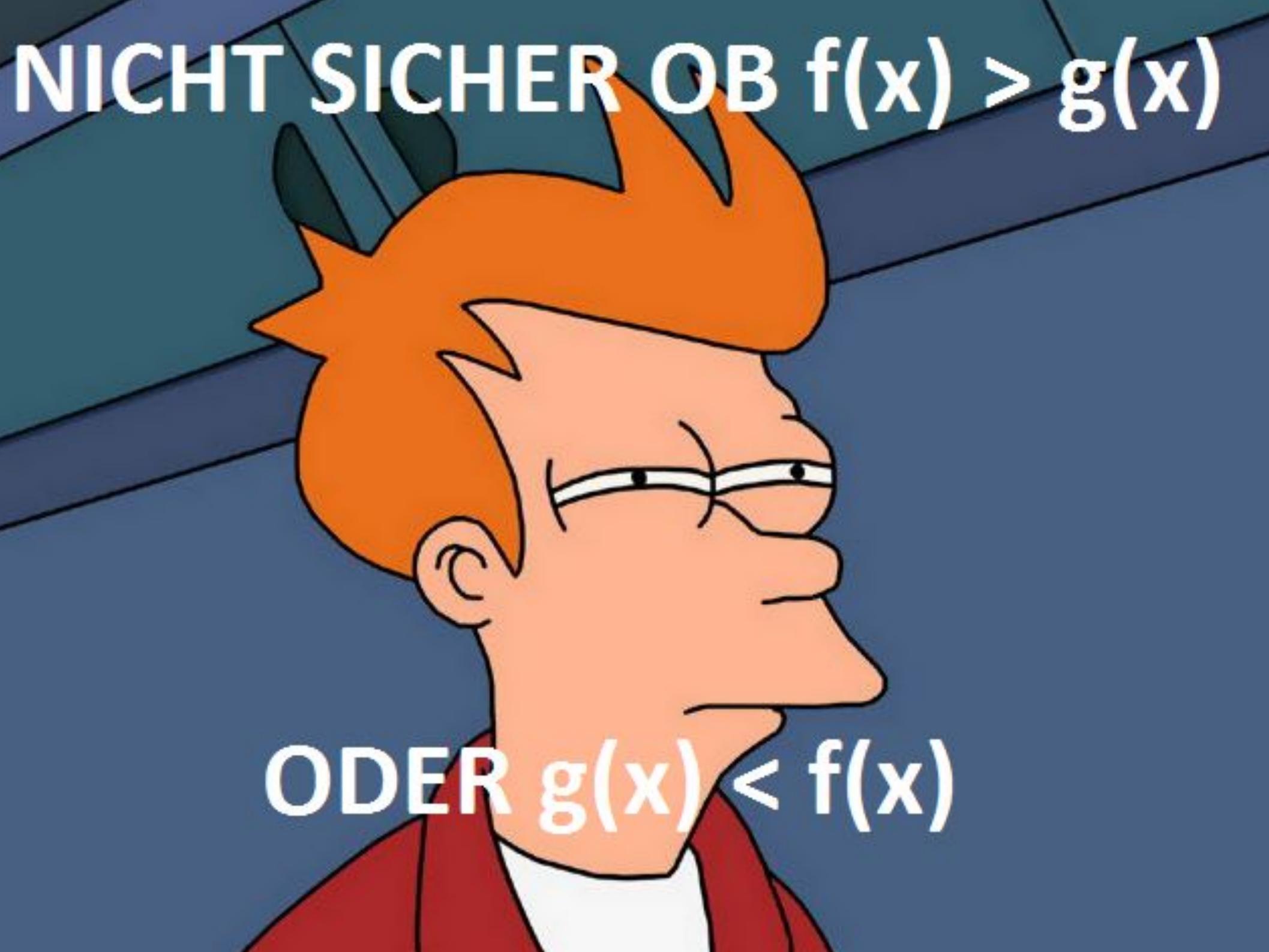
n = Größe des Arrays



Asymptotische Komplexität

- Laufzeit und Speicherverbrauch wird in einer asymptotischen Notation beschrieben, die weitgehend von unwesentlichen Details abstrahiert
- wir machen nur Aussagen über das Verhalten für **sehr große** Eingabegrößen
- vergleich von zwei Komplexitäten über alle natürlichen Zahlen ist nicht ganz einfach

NICHT SICHER OB $f(x) > g(x)$

A cartoon illustration of Homer Simpson from the TV show 'The Simpsons'. He is shown from the chest up, wearing his signature red shirt over a white collared shirt. He has his characteristic orange hair and is looking directly at the viewer with a confused or worried expression, his eyebrows are furrowed and his mouth is slightly open. The background behind him is a dark blue.

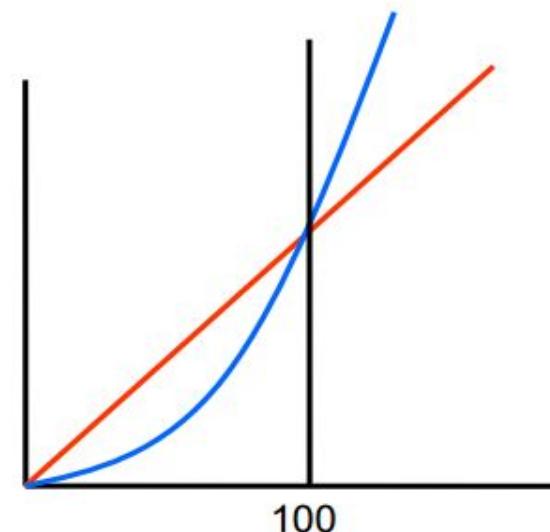
ODER $g(x) < f(x)$

Asymptotische Komplexität

- Vergleich von zwei Komplexität Funktionen über alle natürlichen Zahlen ist nicht ganz einfach
- Wir übernehmen eine mathematische Notation, die zum Vergleichen von Funktionen bis auf einen Faktor verwendet wird

$$T_1(n) = 100 \cdot n$$

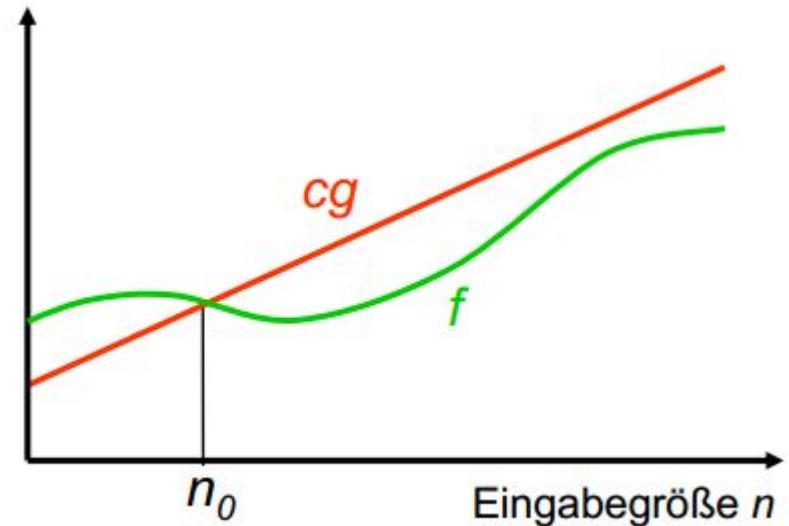
$$T_2(n) = n^2$$



Asymptotische Komplexität

O-Notation: wenn eine Funktion $f(n)$ höchstens so schnell wächst wie eine andere Funktion $g(n)$. $g(n)$ ist also **die obere Schranke** für $f(n)$.

$f(n)$ ist in $\mathcal{O}(g(n))$, wenn es ein $c > 0$ und ein $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \geq n_0$ $f(n) \leq c \cdot g(n)$ gilt





Asymptotische Komplexität

- O-Notation: Abstraktion durch
 - ignorieren endlich vieler Anfangswerte (Spezialfälle) durch
$$n \geq n_0$$
 - Einführung des konstanten Faktors c in der Definition, der von nur durch Konstanten hervorgerufenen Unterschieden abstrahiert
- Beispiel:
 - $T1(n) = 100 * n \in O(n)$: $T1(n)$ wächst höchstens so schnell wie n
 - $T2(n) = n*n \in O(n*n)$: $T2(n)$ wächst höchstens so schnell wie $n*n$



Asymptotische Komplexität

Häufige Größenordnung der Komplexität:

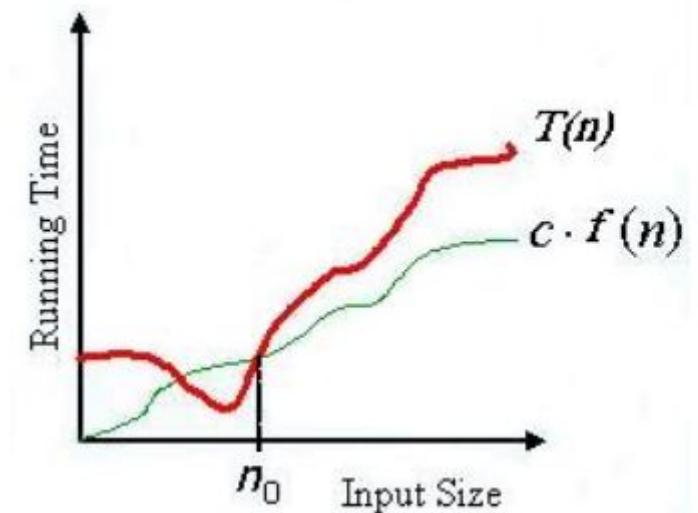
- $O(1)$: In konstanter (von n unabhängiger) Zeit ausführbar
- $O(\log n)$: Bei Verdoppelung von n läuft das Programm um eine konstante Zeit länger
- $O(n)$: Linear Laufzeit proportional zu n
- $O(n^2)$: Quadratische Laufzeit
- $O(n^3)$: Kubische Laufzeit; nur für kleinere n geeignet
- $O(2^n)$: Exponentielles Wachstum; solche Programme sind in der Praxis fast immer wertlos



Asymptotische Komplexität

Ω -Notation: wenn eine Funktion $f(n)$ mindestens so schnell wächst wie eine andere Funktion $g(n)$. $g(n)$ ist also die **untere Schranke** für $f(n)$.

$f(n)$ ist in $\Omega(g(n))$, wenn es ein $c > 0$ und ein $n_0 \in \mathbb{N}$ gibt, so dass für alle $n \geq n_0$ $f(n) \geq c \cdot g(n)$ gilt

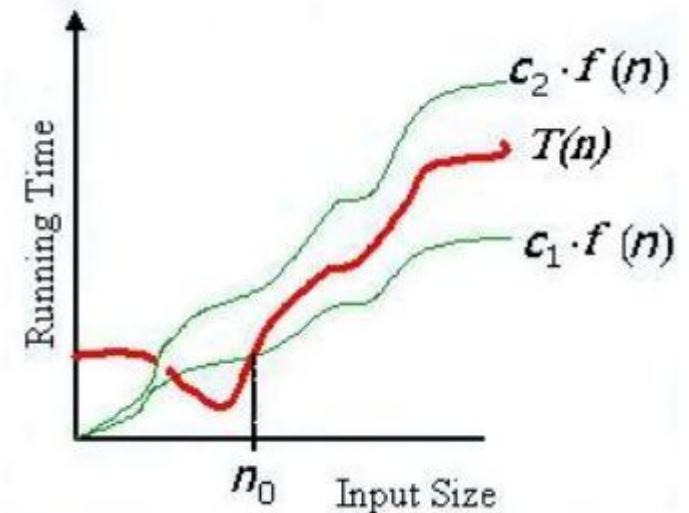




Asymptotische Komplexität

Θ-Notation: wenn eine Funktion $f(n)$ sowohl von oben als auch von unten durch $g(n)$ beschränkt ist. $g(n)$ ist also **die exakte Schranke** für $f(n)$.

$\Theta(g(n))$ ist definiert durch $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.





Beispiele

```
def f1(n):
    s = 0
    for i in range(1,n+1):
        s=s+i
    return s
```

$$T(n) = \sum_{(i=1)}^n 1 = n \rightarrow T(n) \in \Theta(n)$$

Overall complexity $\Theta(n)$

Best/Average/Worst case is the same

```
def f2(n):
    i = 0
    while i<=n:
        #atomic operation
        i = i + 1
```

$$T(n) = \sum_{(i=1)}^n 1 = n \rightarrow T(n) \in \Theta(n)$$

Overall complexity $\Theta(n)$

Best/Average/Worst case is the same

```
def f3(l):
    """
    l - list of numbers
    return True if the list contains
    an even nr
    """
    poz = 0
    while poz<len(l) and l[poz]%2 !=0:
        poz = poz+1
    return poz<len(l)
```

Best case:

The first element is an even number: $T(n)=1 \in \Theta(1)$

Worst case: No even number in the list: $T(n)=n \in \Theta(n)$

Average Case:

While can be executed 1,2,..n times (same probability).

Number of steps = the average number of while iterations

$$T(n) = (1+2+\dots+n)/n = (n+1)/2 \rightarrow T(n) \in \Theta(n)$$

Overall complexity $O(n)$



Suchverfahren

- Verfahren, das in einem Suchraum nach Mustern oder Objekten mit bestimmten Eigenschaften sucht
- **vielfältige Anwendungsbereiche**
 - Suchen in Datenbanken, Google-Search
 - Suchen nach ähnlichen Mustern: z.B. Viren, Malware
 - Bilderkennungsverfahren: Suchen nach Pattern
- **für uns:** einfache Suchverfahren auf Listen



Anforderungen

- statische, kleine Menge, selten Operations notwendig
 - **Lösung:** Feld als Datenstruktur und sequentielles Suchen $\mathcal{O}(n)$
- statische, kleine Menge, häufige Operations
 - **Lösung:** Vorsortiertes $\mathcal{O}(n \log n)$ Feld, binäres Suchen $\mathcal{O}(\log n)$
- dynamisch, große Menge von Elementen
 - **Lösung:** Baum als dynamische Datenstruktur, organisiert als binärer Suchbaum $\mathcal{O}(h)$, h ist Baumhöhe. Worst-Case: $h = n$, Best-Case: $h = \log n$ (**balanciert**)
- dynamisch, große Menge, viele, effiziente Zugriffe notwendig
 - **Lösung:** Binärer Suchbaum, der eine möglichst geringe Höhe h garantiert: z.B. **B-Baum**



Charakteristiken

- Eingabe: Folge von Zahlen $a, n, \langle a_1, a_2, \dots, a_n \rangle$
 - Vorbedingung: $n \in \mathbb{N}, n \geq 0;$
- Ausgabe: p
 - Nachbedingungen : $(0 \leq p \leq n-1 \text{ and } k = a[p]) \text{ or } (p=-1)$

In der Praxis:

- gespeichert in Arrays, Linked Lists, ...
- Charakterisierung der gesuchten Objekte durch Such-Schlüssel
- Such-Schlüssel können z.B. Attribute der Objekte sein
- Beispiel: **ID von Personen**



Einfache Suchverfahren

- aufwand für alle Verfahren etwa gleich groß
 - außer Linearem Suchen
- einfachstes Suchverfahren verwenden
 - binäres Suchen
- exponentielles Suchen
 - bei ausgelagerten Daten



Lineare Suche

Gegeben sei $A[1 \dots n]$ und k (ein Schlüssel)

Idee der linearen Suche:

- sequentielles Durchlaufen des Feldes A
- Vergleich der Schlüssel $A[i]$, $i=1, \dots, n$ mit dem Such-Schlüssel k

```
def searchSeq(el, l):
    """
        Search for an element in a list
        el - element
        l - list of elements
        return the position of the element
            or -1 if the element is not in l
    """
    poz = -1
    for i in range(0, len(l)):
        if el==l[i]:
            poz = i
    return poz
```

```
def searchSucc(el, l):
    """
        Search for an element in a list
        el - element
        l - list of elements
        return the position of first occurrence
            or -1 if the element is not in l
    """
    i = 0
    while i<len(l) and el!=l[i]:
        i=i+1
    if i<len(l):
        return i
    return -1
```



Laufzeitanalyse

- Best-Case: sofortiger Treffer: $T(n) = 1$, also $T(n) = O(1)$
- Worst-Case: alles durchsuchen: $T(n) = n$, also $T(n) = O(n)$
- Average-Case: erfolgreiche Suche unter der Annahme, dass jede Anordnung der Elemente gleich wahrscheinlich ist:

$$T(n) = (1+2+\dots+n-1)/n \text{ also } T(n) = O(n)$$



Sortiertes Feld

```
def searchSeq(el,l):
    """
        Search for an element in a list
        el - element
        l - list of ordered elements
        return the position of first occurrence
            or the position where the element
            can be inserted
    """
    if len(l)==0:
        return 0
    poz = -1
    for i in range(0,len(l)):
        if el<=l[i]:
            poz = i
    if poz===-1:
        return len(l)
    return poz
```

```
def searchSucc(el,l):
    """
        Search for an element in a list
        el - element
        l - list of ordered elements
        return the position of first occurrence
            or the position where the element
            can be inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    i = 0
    while i<len(l) and el>l[i]:
        i=i+1
    return i
```



Fazit

- sehr einfaches Verfahren
- eignet sich auch für einfach verkettete Listen
- das Verfahren ist auch für unsortierte Felder geeignet
- aber das Verfahren ist nur für kleine Werte von n **praktikable**



Binäre Suche

- Falls in einer Folge häufig gesucht werden muss, so lohnt es sich, die Feldelemente sortiert zu speichern
- **Eingabe:** Sortiertes Feld
 - halbieren des Suchraums in jedem Schritt, indem **der gesuchte Wert mit dem Wert auf der Mittelposition des geordneten Feldes** verglichen wird
 - gesuchter Wert ist kleiner: weiterarbeiten mit linkem Teilstück
 - gesuchter Wert ist größer: weiterarbeiten mit rechtem Teilstück



Laufzeitanalyse

- **Zählen der Anzahl der Vergleiche**
- Best-Case:
 - sofortiger Treffer: $T(n) = 1$, also $T(n) = O(1)$
- Worst-Case:
 - Suchraum muss solange halbiert werden, bis er nur noch 1 Element enthält,
 - oft logarithmisch
 - $T(n) = T(n/2) + 1 = \log(n + 1)$, $T(n) = O(\log n)$



Binäre Suche

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

Find: 37

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

↑
first ↑
middle ↑
last

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

↑
first ↑
middle ↑
last

Find: 37

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

↑
middle ↑
first ↑
last

0	1	2	3	4	5	6	7	8
20	35	37	40	45	50	51	55	67

↑
middle



Binäre Suche





Binäre Suche

```
def searchBinaryNonRec(el, l):
    """
        Search an element in a list
        el - element to be searched
        l - a list of ordered elements
        return the position of first occurrence or the position where the element can be
    inserted
    """
    if len(l)==0:
        return 0
    if el<=l[0]:
        return 0
    if el>=l[len(l)-1]:
        return len(l)
    right=len(l)
    left = 0
    while right-left>1:
        m = (left+right)/2
        if el<=l[m]:
            right=m
        else:
            left=m
    return right
```



Fazit

- gut geeignet für große Werte n
- Beispiel:
 - sei $n = 2$ Millionen
 - Lineare Suche benötigt im Worst Case 2 Millionen Vergleiche
 - Binäre Suche benötigt: **log (2 * 10^6) ~ 20** Vergleiche
- nicht gut geeignet, wenn sich die Daten häufig ändern



In Python

index()

```
l = range(1,10)
try:
    poz = l.index(11)
except ValueError:
    # element is not in the list
```

`__eq__, __gt__, __lt__, ... __cmp__`

```
class MyClass:
    def __init__(self,id,name):
        self.id = id
        self.name = name

    def __eq__(self,ot):
        return self.id == ot.id

#    def __cmp__(self,ot):
#        return self.id.__cmp__(ot.id)
```

```
def testIndex():
    l = []
    for i in range(0,200):
        ob = MyClass(i,"ad")
        l.append(ob)

    findObj = MyClass(32,"ad")
    print "positions:" +str(l.index(findObj))
```



In Python

in

```
l = range(1,10)
found = 4 in l
```

__iter__, next

```
class MyClass2:
    def __init__(self):
        self.l = []

    def add(self,obj):
        self.l.append(obj)

    def __iter__(self):
        """
            Return an iterator object
        """
        self.iterPoz = 0
        return self
```

```
def next(self):
    """
        Return the next element in the iteration
        raise StopIteration exception if we are at the end
    """
    if (self.iterPoz>=len(self.l)):
        raise StopIteration()

    rez = self.l[self.iterPoz]
    self.iterPoz = self.iterPoz +1
    return rez

def testIn():
    container = MyClass2()
    for i in range(0,200):
        container.add(MyClass(i, "ad"))
    findObj = MyClass(20, "asdasd")
    print findObj in container
```



In Python

```

def measureBinary(e, l):
    sw = StopWatch()
    poz = searchBinaryRec(e, l)
    print "    BinaryRec in %f sec; poz=%i" %(sw.stop(),poz)

def measurePythonIndex(e, l):
    sw = StopWatch()
    poz = -2
    try:
        poz = l.index(e)
    except ValueError:
        pass #we ignore the error..
    print "    PythIndex in %f sec; poz=%i" %(sw.stop(),poz)

def measureSearchSeq(e, l):
    sw = StopWatch()
    poz = searchSeq(e, l)
    print "    searchSeq in %f sec; poz=%i" %(sw.stop(),poz)

```

search 200	search 10000000
BinaryRec in 0.000000 sec; poz=200	BinaryRec in 0.000000 sec; poz=10000000
PythIndex in 0.000000 sec; poz=200	PythIndex in 0.234000 sec; poz=10000000
PythonIn in 0.000000 sec	PythonIn in 0.238000 sec
BinaryNon in 0.000000 sec; poz=200	BinaryNon in 0.000000 sec; poz=10000000
searchSuc in 0.000000 sec; poz=200	searchSuc in 2.050000 sec; poz=10000000



Sortierproblem

- **Eingabe:** Folge von Zahlen $a, n, \langle a_1, a_2, \dots, a_n \rangle$
- **Ausgabe:** sortierte Folge der Eingabe $\langle a'_1, a'_2, \dots, a'_n \rangle$ mit $a'_1 \leq a'_2 \leq \dots \leq a'_n$
- Eingabemenge als Feld oder verkettete Liste repräsentiert
- Sortierverfahren lösen das durch die Eingabe-Ausgabe-Relation beschriebene Sortierproblem



Struktur der Daten

- zu sortierende Werte (Schlüssel) sind selten isoliert
 - sondern Teil einer größeren Datenmenge (Datensatz, Record)
- Daten bestehen aus Schlüssel und Satellitendaten
- Satellitendaten werden mit Schlüssel umsortiert
- im Folgenden werden Satellitendaten ignoriert



Eigenschaften

- Effizienz
 - Best-, Average-, Worst-Case
- Speicherbedarf
 - in-place (zusätzlicher Speicher von der Eingabegröße unabhängig)
 - out-of-place
- rekursiv oder iterativ
- Stabilität
 - stabile Verfahren verändern die Reihenfolge von äquivalenten Elementen nicht
- verwendete Operationen
 - Vertauschen, Auswählen, Einfügen
- Verwendung spezieller Datenstrukturen



Sortieren von Spielkarten

- **Bubble Sort**
 - Aufnehmen aller Karten vom Tisch
 - vertausche ggf. benachbarte Karten, bis Reihenfolge korrekt
- **Selection Sort**
 - Aufnehmen der jeweils niedrigsten Karte vom Tisch
 - Anfügen der Karte am Ende
- **Insertion Sort**
 - Aufnehmen einer beliebigen Karte
 - Einfügen der Karte an der korrekten Position



Bubble-Sort

- durchlaufe die Menge
- vertausche zwei aufeinanderfolgende Elemente, wenn ihre Reihenfolge nicht stimmt
- durchlaufe die Menge gegebenenfalls mehrmals, bis bei einem Durchlauf keine Vertauschungen mehr durchgeführt werden mussten



Bubble-Sort

					sortiert = true
55	7	78	12	42	55<7? tausche(7,55); sortiert = false;
7	55	78	12	42	55<78?
7	55	78	12	42	78<12? tausche(78,12);
7	55	12	78	42	78<42? tausche(78,42); Ende: sortiert? sortiert=true;
7	55	12	42	78	7<55?
7	55	12	42	78	55<12? tausche(55,12); sortiert=false;
7	12	55	42	78	55<42? tausche(55,42);
7	12	42	55	78	55<78? Ende: sortiert? sortiert=true;
7	12	42	55	78	7<12?
7	12	42	55	78	12<42?
7	12	42	55	78	42<55?
7	12	42	55	78	55<78? Ende: sortiert? Fertig.



Python





Eigenschaften

- iterativ
- stabil
 - (gleiche benachbarte Schlüssel werden nicht getauscht)
- in-place
 - (konstanter zusätzlicher Speicheraufwand)
- effizient für vorsortierte Mengen



Laufzeit

- schlechtester Fall
 - Eingabe ist umgekehrt sortiert: $n, n-1, \dots, 2, 1$
 - $n-1$ Vertauschungen im ersten Durchlauf
 - $n-2$ Vertauschungen im zweiten Durchlauf
 - ...
 - 1 Vertauschung im n -ten Durchlauf

$$T(n) = n(n-1)/2 \in O(n^2)$$

- bester Fall
 - Eingabe ist sortiert: $1, 2, \dots, n-1, n$
 - ein Durchlauf $O(n)$



Selection-Sort

- durchlaufe die Menge, finde das kleinste Element
- vertausche das kleinste Element mit dem ersten Element
- vorderer Teil ist sortiert (k Elemente nach Durchlauf k), hinterer Teil ist unsortiert ($n-k$ Elemente)
- durchlaufe die hintere, nicht sortierte Teilmenge, finde das n -kleinste Element
- vertausche das n -kleinste Element mit dem n -ten Element



Selection-Sort

a[1] a[2] a[3] a[4] a[5]

55	7	78	12	42
7	55	78	12	42
7	12	78	55	42
7	12	42	55	78

1. Durchlauf: $7 = \min(1..n)$; tausche 7 mit a[1];
2. Durchlauf: $12 = \min(2..n)$; tausche 12 mit a[2];
3. Durchlauf: $42 = \min(3..n)$; tausche 42 mit a[3];
4. Durchlauf: $55 = \min(4..n)$; tausche 55 mit a[4];



Python





Eigenschaften

- iterativ
- instabil
 - gleiche benachbarte Schlüssel werden getauscht
 - lässt sich auch stabil implementieren
- in-place
 - konstanter zusätzlicher Speicheraufwand



Laufzeit

- bester, mittlerer, schlechtester Fall
 - für n Einträge werden $n-1$ Minima gesucht
 - $n-1$ Vergleiche für erstes Minimum
 - $n-2$ Vergleiche für zweites Minimum
 - ...
 - 1 Vergleich für Minimum $n-1$
 - $T(n) = n(n-1)/2 \in O(n^2)$



Insertion-Sort

- erstes Element ist sortiert, hinterer Teil mit $n-1$ Elementen ist unsortiert
- entnehme der hinteren, unsortierten Menge ein Element und füge es an die richtige Position der vorderen, sortierten Menge ein ($n-1$ mal)
- Einfügen in die vordere, sortierte Menge erfordert das Verschieben von Elementen



Insertion-Sort

5	2	4	6	1	3
---	---	---	---	---	---

Elemente 1..1 sind sortiert, 2..n unsortiert

5	2	4	6	1	3
---	---	---	---	---	---

Vergleiche 2 mit allen Elementen der sortierten Menge beginnend mit dem größten. Wenn ein Element größer als 2 ist, schiebe es eins nach rechts, sonst füge 2 ein.

2	5	4	6	1	3
---	---	---	---	---	---

Elemente 1..2 sind sortiert, 3..n unsortiert

2	5	4	6	1	3
---	---	---	---	---	---

Vergleiche 4 mit allen Elementen der sortierten Menge. Wenn ein Element größer als 4 ist, verschiebe es.

2	4	5	6	1	3
---	---	---	---	---	---

Elemente 1..3 sind sortiert. Einfügen von 6.

2	4	5	6	1	3
---	---	---	---	---	---

Elemente 1..4 sind sortiert. Einfügen von 1 (Dazu werden Elemente 6,5,4 und 2 jeweils um eins nach rechts verschoben. Danach wird 1 an Pos. eins eingefügt.)

1	2	4	5	6	3
---	---	---	---	---	---

...

1	2	3	4	5	6
---	---	---	---	---	---



Insertion-Sort





Eigenschaften

- iterativ
- stabil
 - gleiche benachbarte Schlüssel werden nicht getauscht
- in-place
 - (konstanter zusätzlicher Speicheraufwand)
- effizient für vorsortierte Mengen



Laufzeit

- **bester Fall**
 - Menge ist vorsortiert
 - innere while-Schleife wird nicht durchlaufen
 - $O(n)$

- **schlechtester Fall**
 - Menge ist umgekehrt sortiert
 - $k-1$ Verschiebeoperationen für das k -te Element
 - $O(n^2)$

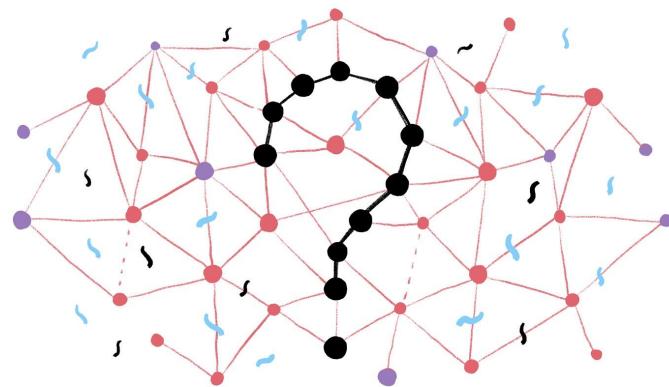


Nächste Woche

- weitere Algorithmen
- Quicksort
- Mergesort
- Counting
- Bucket



Algorithmen und Python





Problemlösen

- Problem definieren
- Algorithmus entwerfen
- Programm erstellen
- Lösung überprüfen





Algorithmenentwurf

- entwurf neuer Algorithmen erfordert Kreativität
- allgemeine Denk- und Lösungsansätze können dabei oft sehr nützlich sein:
 - liefern uns Muster zur Lösung unterschiedlicher Probleme
 - können systematisch in gängige Kontroll- und Datenstrukturen diverser Programmiersprachen umgesetzt werden
 - das Verhalten (**Effizienz, Komplexität**) der entsprechenden Algorithmen ist bekannt

Techniken zum Entwurf

- Divide et Impera
- Greedy
- Dynamische Programmierung





Teile und herrsche

- rekursive Anwendung eines Algorithmus auf immer kleiner Teilprobleme
- direkte Lösung eines hinreichend kleinen Teilproblems
 - häufig Datenmenge in Teilmengen zerlegen
 - Rekursion
- **teile** das Gesamtproblem in kleinere Teilprobleme auf
- **herrsche** über die Teilprobleme durch rekursives Lösen
 - wenn die Teilprobleme klein sind, löse sie direkt
- **verbinde** die Lösungen der Teilprobleme zur Lösung des Gesamtproblems



Prinzip

Methode V zur Lösung des Problems P der Größe n

Methode $V(P)$

wenn $n < d$

dann

löse das Problem P direkt;

sonst

teile P in mehrere Teilprobleme

P_1, P_2, \dots, P_k mit $k \geq 2$;

Methode $V(P_1)$; ...; Methode $V(P_k)$;

verbinde die Teillösungen für

P_1, P_2, \dots, P_k zur Gesamtlösung für P ;



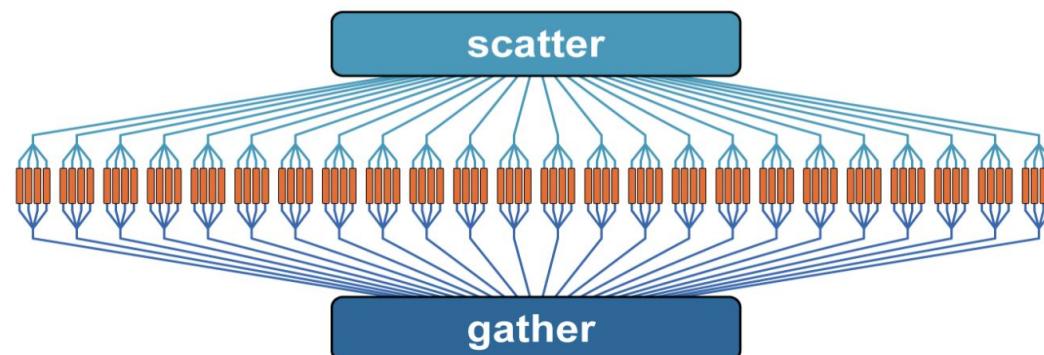


Eigenschaften

- ***kann bei konzeptionell schwierigen Problemen hilfreich sein***
- Lösung für den Trivialfall muss bekannt sein
- Aufteilung in Teilprobleme muss möglich sein
- Kombination der Teillösungen muss möglich sein

Eigenschaften

- kann effiziente Lösungen realisieren
 - wenn
 - die Lösung des Trivial Falls in $\mathcal{O}(1)$ liegt
 - Aufteilung in Teilprobleme und Kombination der Teillösungen in $\mathcal{O}(n)$ liegt
 - und die Zahl der Teilprobleme beschränkt ist
 - liegt der Algorithmus in $\mathcal{O}(n^* \log(n))$
- für Parallelverarbeitung geeignet
 - Teilprobleme werden unabhängig voneinander verarbeitet





Implementierung

Definition des Trivialfalls

- möglichst kleine direkt zu lösende Teilprobleme sind elegant und einfach
- Effizienz wird verbessert, wenn schon relativ große Teilprobleme direkt gelöst werden
- Rekursionstiefe sollte nicht zu groß werden



Implementierung

Aufteilung in Teilprobleme

- Wahl der Anzahl der Teilprobleme und konkrete Aufteilung kann anspruchsvoll sein

Kombination der Teillösungen

- typischerweise konzeptionell anspruchsvoll





Komplexitätsanalyse

Aufteilen des Problems bei allgemeinem **divide et impera** in Teilprobleme der Größe $[n/b]$, wobei $g(n)$ die Kosten des Aufteilens und $h(n)$ die Kosten des Zusammenfügens der Lösungen beschreibt.

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + \underbrace{g(n) + h(n)}_{f(n)} = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(n) \in \Theta\left(n^{\log_b a} \log n\right) \text{ falls } f(n) \in \Theta\left(n^{\log_b a}\right)$$



Maximum in einem Array

- Das Maximum für Arrays der Größe 1 wird sofort zurückgegeben
- Wenn $N > 1$, wird das Array geteilt und das Maximum für beide Teile zurückgegeben



Maximum in einem Array





Maximum in einem Array

Recurrence: $T(n) = \begin{cases} 1 & \text{for } n=1 \\ T(n-1)+1 & \text{otherwise} \end{cases}$

$$\begin{aligned} T(n) &= T(n-1)+1 \\ T(n-1) &= T(n-2)+1 \\ T(n-2) &= T(n-3)+1 \Rightarrow T(n) &= 1+1+\dots+1 = n \in \Theta(n) \\ \dots &= \dots \\ T(2) &= T(1)+1 \end{aligned}$$

Recurrence: $T(n) = \begin{cases} 1 & \text{for } n=1 \\ 2T(n/2)+1 & \text{otherwise} \end{cases}$

$$\begin{aligned} T(2^k) &= 2T(2^{(k-1)})+1 \\ 2T(2^{(k-1)}) &= 2^2T(2^{(k-2)})+2 \end{aligned}$$

Denote: $n=2^k \Rightarrow k=\log_2 n$ $2^2T(2^{(k-2)})=2^3T(2^{(k-3)})+2^2 \Rightarrow$

$$\dots = \dots \\ 2^{(k-1)}T(2) = 2^kT(1) + 2^{(k-1)}$$

$$T(n) = 1 + 2^1 + 2^2 + \dots + 2^k = (2^{(k+1)} - 1)/(2-1) = 2^k 2 - 1 = 2n - 1 \in \Theta(n)$$



`pow(x, k)` , $k > 1$

- For-Schleife: $\text{pow}(x, k) = x * \dots * x$

- Divide et impera

$$x^k = \begin{cases} x^{(k/2)} x^{(k/2)} & \text{for } k \text{ even} \\ x^{(k/2)} x^{(k/2)} x & \text{for } k \text{ odd} \end{cases}$$

```
def power(x, k):
    """
        compute x^k
        x real number
        k integer number
        return x^k
    """
    if k==1:
        #base case
        return x
    #divide
    half = k/2
    aux = power(x, half)
    #conquer
    if k%2==0:
        return aux*aux
    else:
        return aux*aux*x
```

Divide: compute $k/2$

Conquer: 1 recursive call to compute $x^{(k/2)}$

Combine: 1 ore 2 multiplications

Time complexity: $T(n) \in \Theta(\log_2 n)$



Gierige Algorithmen

- eine Algorithmenmethode, um **Optimierungsprobleme** zu lösen
- bei einem Optimierungsproblem gibt es zu jeder Probleminstanz **viele mögliche oder zulässige Lösungen**
- Lösungen haben Werte gegeben durch eine Zielfunktion
- gesucht ist dann **eine möglichst gute zulässige Lösung**
 - also eine Lösung mit möglichst kleinem oder möglichst großem Wert



Gierige Algorithmen Bestandteile

- eine sich verbrauchende **Kandidatenliste**
- eine sich aufbauende **Resultatliste**
- eine **SelectMostPromising-Funktion**, die aus der Kandidatenliste den jeweils besten Teil auswählt
- eine **Solution-funktion**, die prüft, ob die aktuelle Resultatliste eine Lösung des Problems darstellt
- eine **Acceptable-Funktion**, die prüft, ob die Erweiterung einer Resultatliste durch den nächsten Kandidaten zulässig ist
- eine **Zielfunktion**, die den Wert einer Lösung angibt



Python

```
def greedy(c):
    """
        Greedy algorithm
        c - a list of candidates
        return a list (B) the solution found (if exists) using the greedy
        strategy, None if the algorithm
        selectMostPromissing - a function that return the most promising
        candidate
        acceptable - a function that returns True if a candidate solution can be
        extended to a solution
        solution - verify if a given candidate is a solution
    """
    b = [] #start with an empty set as a candidate solution
    while not solution(b) and c!=[]:
        #select the local optimum (the best candidate)
        candidate = selectMostPromissing(c)
        #remove the current candidate
        c.remove(candidate)
        #if the new extended candidate solution is acceptable
        if acceptable(b+[candidate]):
            b.append(candidate)

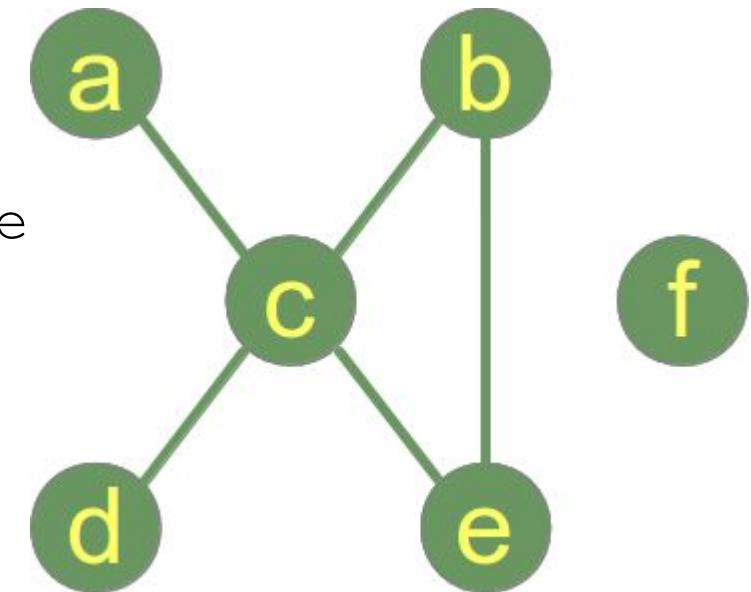
    if solution(b):
        return b
    #there is no solution
    return None
```



Graphen in Python

- ein Graph besteht aus Knoten und Kanten
 - Knoten können miteinander verbunden sein
 - diese Verbindung zwischen zwei Knoten nennt man Kante

```
graph = { "a" : ["c"] ,  
          "b" : ["c", "e"] ,  
          "c" : ["a", "b", "d", "e"] ,  
          "d" : ["c"] ,  
          "e" : ["c", "b"] ,  
          "f" : []  
}
```





Graphen in Python

```
graph = { "a" : ["c"],
          "b" : ["c", "e"],
          "c" : ["a", "b", "d", "e"],
          "d" : ["c"],
          "e" : ["c", "b"],
          "f" : []}
```

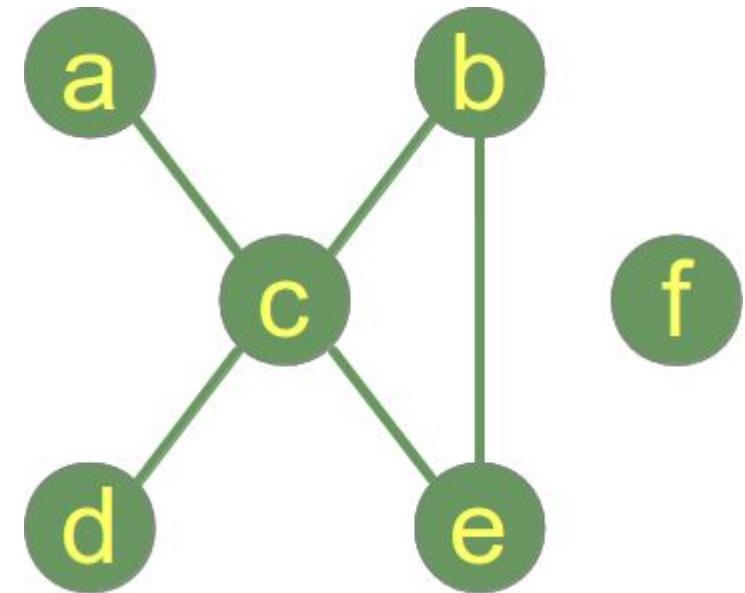
```
def generate_edges(graph):
    edges = []
    for node in graph:
        for neighbour in graph[node]:
            edges.append((node, neighbour))

    return edges

print(generate_edges(graph))
```

Output:

```
[('a', 'c'), ('c', 'a'), ('c', 'b'), ('c', 'd'),
 ('c', 'e'), ('b', 'c'), ('b', 'e'), ('e', 'c'),
 ('e', 'b'), ('d', 'c')]
```





mit Klassen (<https://www.python-kurs.eu/>)

```

class Graph(object):
    def __init__(self, graph_dict={}):
        """ initializes a graph object """
        self.__graph_dict = graph_dict

    def vertices(self):
        """ returns the vertices of a graph """
        return list(self.__graph_dict.keys())

    def edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()

    def add_vertex(self, vertex):
        """ If the vertex "vertex" is not in
            self.__graph_dict, a key "vertex" with an empty
            list as a value is added to the dictionary.
            Otherwise nothing has to be done.
        """
        if vertex not in self.__graph_dict:
            self.__graph_dict[vertex] = []

    def add_edge(self, edge):
        """ assumes that edge is of type set, tuple or list;
            between two vertices can be multiple edges!
        """

        edge = set(edge)
        (vertex1, vertex2) = tuple(edge)
        if vertex1 in self.__graph_dict:
            self.__graph_dict[vertex1].append(vertex2)
        else:
            self.__graph_dict[vertex1] = [vertex2]

```

```

def __generate_edges(self):
    """ A static method generating the edges of the
        graph "graph". Edges are represented as sets
        with one (a loop back to the vertex) or two
        vertices
    """
    edges = []
    for vertex in self.__graph_dict:
        for neighbour in self.__graph_dict[vertex]:
            if {neighbour, vertex} not in edges:
                edges.append({vertex, neighbour})
    return edges

def __str__(self):
    res = "vertices: "
    for k in self.__graph_dict:
        res += str(k) + " "
    res += "\nedges: "
    for edge in self.__generate_edges():
        res += str(edge) + " "
    return res

```

```

if __name__ == "__main__":
    g = { "a" : ["d"],
          "b" : ["c"],
          "c" : ["b", "c", "d", "e"],
          "d" : ["a", "c"],
          "e" : ["c"],
          "f" : []
        }

    graph = Graph(g)

    print("Vertices of graph:")
    print(graph.vertices())

    print("Edges of graph:")
    print(graph.edges())

    print("Add vertex:")
    graph.add_vertex("z")

    print("Vertices of graph:")
    print(graph.vertices())

    print("Add an edge:")
    graph.add_edge({"a","z"})

    print("Vertices of graph:")
    print(graph.vertices())

    print("Edges of graph:")
    print(graph.edges())

    print('Adding an edge {"x","y"} with new vertices:')
    graph.add_edge({"x","y"})
    print("Vertices of graph:")
    print(graph.vertices())
    print("Edges of graph:")
    print(graph.edges())

```



Minimale Spannbäume

- Input: gewichteter, ungerichteteter Graph
- Zulässige Lösungen: Spannbäume
 - ein Baum
 - welcher alle Knoten dieses Graphen enthält
- Zielfunktion: Gewicht eines Spannbaums
- Output: Spannbaum minimalen Gewichts



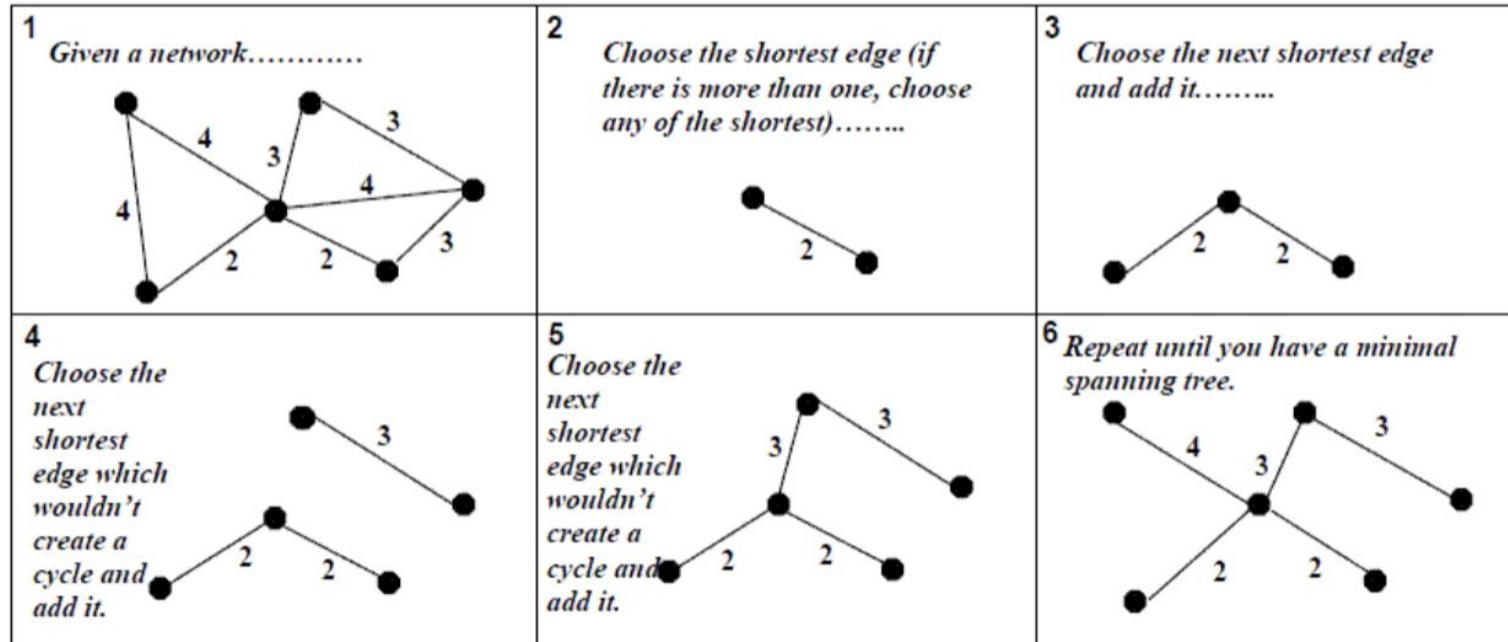
Idee

- Greedy Algorithmen bestimmen die Lösung durch sukzessive Erweiterungen von bereits gefundenen Teillösungen
- erweitern geschieht durch **lokal optimale Wahlen**
- können wir zeigen, dass lokale optimale Wahlen zu global optimalen Lösungen führt?



- Algorithmus von Kruskal
- bestimmt den minimalen Spannbaum durch sukzessives **Hinzufügen von Kanten**
- wählt die möglichst leichte Kante aus, die zu **keinem Zyklus** führt
- formale Analyse zeigt, dass Teilbaum immer in einem minimalen Spannbaum enthalten ist (induction)

Kruskal



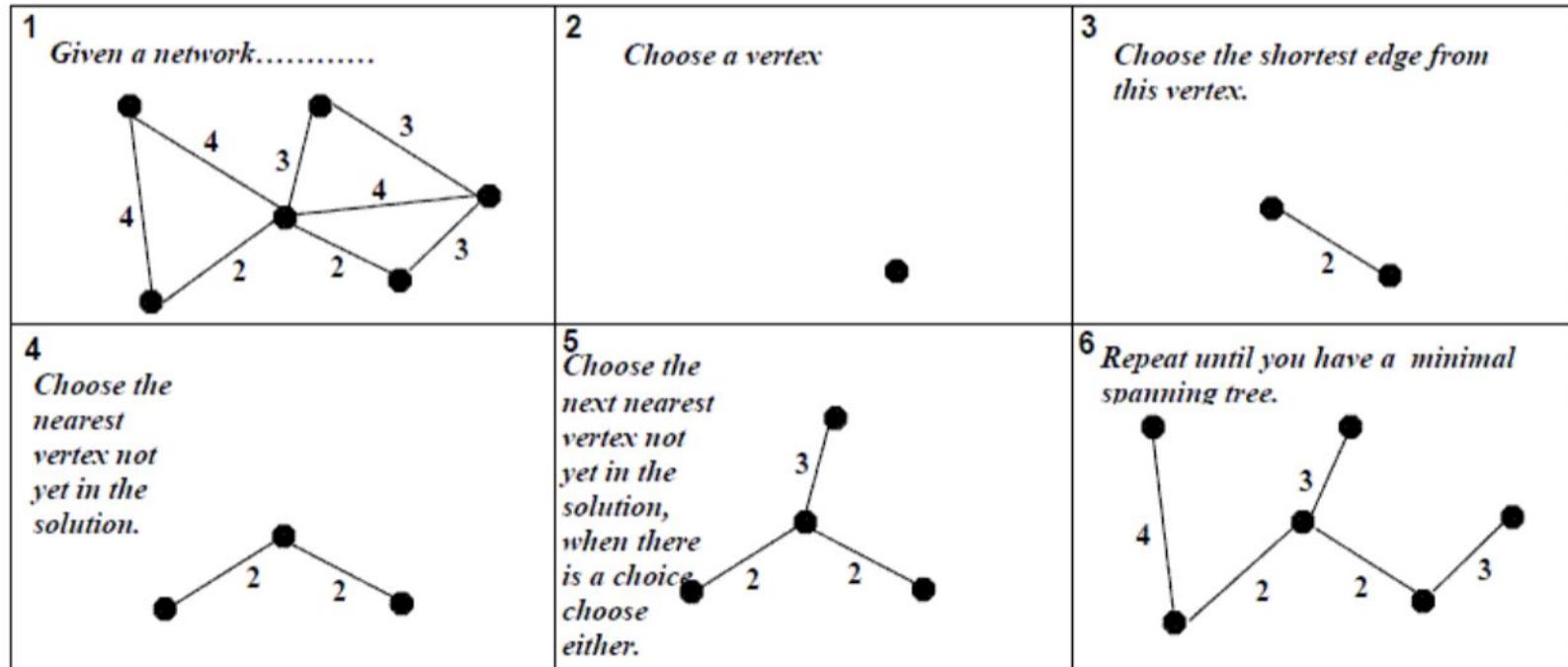
```
adjacency matrix r = [
    [0, 4, 4, 0, 0, 0],
    [4, 0, 2, 0, 0, 0],
    [4, 2, 0, 3, 2, 4],
    [0, 0, 3, 0, 0, 3],
    [0, 0, 2, 0, 0, 3],
    [0, 0, 4, 3, 3, 0]
]
```



Algorithmus von Prim

- bestimmt minimalen Spannbaum durch sukzessives **Hinzufügen von Kanten**
- wählt möglichst leichte Kante aus, die **isolierten Knoten** mit dem **Teilbaum verbindet**
- formale Analyse zeigt, dass Teilbaum immer in einem minimalen Spannbaum enthalten ist

Prim



```
adjacency matrix r = [
    [0, 4, 4, 0, 0, 0],
    [4, 0, 2, 0, 0, 0],
    [4, 2, 0, 3, 2, 4],
    [0, 0, 3, 0, 0, 3],
    [0, 0, 2, 0, 0, 3],
    [0, 0, 4, 3, 3, 0]
]
```

Prim





1-Prozessor-Scheduling

- Gegeben sind n Jobs J_1, \dots, J_n mit Dauer t_1, \dots, t_n
- jeder der Jobs muss auf einem einzigen Prozessor abgearbeitet werden
- der Prozessor kann zu jedem Zeitpunkt nur einen Job bearbeiten
- ein einmal begonnener Job darf nicht abgebrochen werden
- gesucht ist eine Reihenfolge, in der die Jobs abgearbeitet werden, so dass die durchschnittliche Bearbeitungszeit der Jobs möglichst gering ist



1-Prozessor-Scheduling

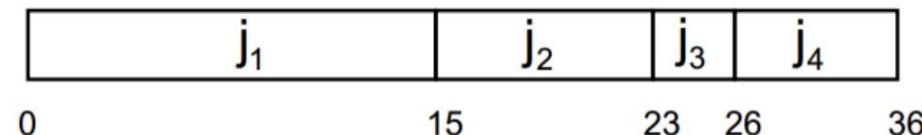
- Bearbeitungszeit eines Jobs ist der Zeitpunkt, an dem der Job vollständig bearbeitet wurde
- werden die Jobs in Reihenfolge $j_{\pi}(1), \dots, j_{\pi}(n)$ ausgeführt, wobei π eine Permutation ist,
- so ist die Bearbeitungszeit von Job $j_{\pi}(1)$ genau $t_{\pi}(1)$, die Bearbeitungszeit von Job $j_{\pi}(2)$ ist $t_{\pi}(1) + t_{\pi}(2)$,
- usw



1-Prozessor-Scheduling

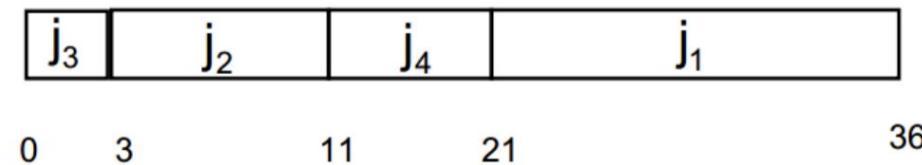
Job	Laufzeit
j ₁	15
j ₂	8
j ₃	3
j ₄	10

Schedule Nr. 1:



Durchschnittliche Beendigung: 25

Schedule Nr. 2:



Durchschnittliche Beendigung: 17,75



1-Prozessor-Scheduling

- Bei Permutation π ist die durchschnittliche Bearbeitungszeit gegeben durch

$$\frac{1}{n} \sum_{i=1}^n (n - i + 1) t_{\pi(i)}$$

- Eine Permutation π führt genau dann zu einer minimalen durchschnittlichen Bearbeitungszeit, wenn die Folge $(t_{\pi(1)}, \dots, t_{\pi(n)})$ aufsteigend sortiert ist



Multi-Prozessor-Scheduling

- Gegeben sind n Jobs J_1, \dots, J_n mit Dauer t_1, \dots, t_n und m identische Prozessoren
- jeder der Jobs muss auf einem einzigen Prozessor abgearbeitet werden
- der Prozessor kann zu jedem Zeitpunkt nur einen Job bearbeiten
- ein einmal begonnener Job darf nicht abgebrochen werden
- gesucht ist eine Reihenfolge, in der die Jobs abgearbeitet werden, so dass die durchschnittliche Bearbeitungszeit der Jobs möglichst gering ist



Multi-Prozessor-Scheduling

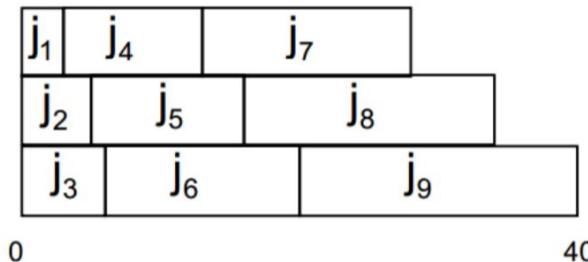
- die Permutation π sei so gewählt, dass die Folge $(t_{\pi}(1), \dots, t_{\pi}(n))$ aufsteigend sortiert ist
- weiter werde dann Job $j_{\pi}(i)$ auf dem Prozessor mit Nummer $i \bmod m$ ausgeführt
- das so konstruiert Scheduling minimiert dann die durchschnittliche Bearbeitungszeit



Multi-Prozessor-Scheduling

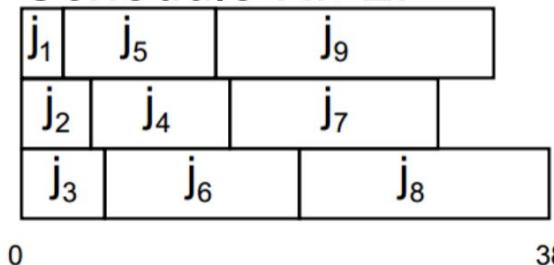
Job	Laufzeit
j ₁	3
j ₂	5
j ₃	6
j ₄	10
j ₅	11
j ₆	14
j ₇	15
j ₈	18
j ₉	20

Schedule Nr. 1:



Durchschnittliche Beendigung: 18,33

Schedule Nr. 2:



Durchschnittliche Beendigung: 18,33



Dynamische Programmierung

- **dynamisch** = sequentiell
- **Programmierung** = Optimierung mit Nebenbedingungen
- die optimale Lösung eines Problems mit Größe n setzt sich **aus optimalen Teillösungen** kleinerer Probleme zusammen



Dynamische Programmierung

- kleinere Teilprobleme mit zunehmender Größe definieren
- diese Teilprobleme in eine Hierarchie gliedern, so dass die optimale Lösung für ein Problem der Größe n aus den optimalen Lösungen kleinerer Teilprobleme zusammengesetzt werden kann
- diese Zusammensetzung wird typischerweise beschrieben durch eine Rekursion
- Ein DP-Algorithmus umfasst folgende Schritte:
 - Teilprobleme bearbeiten
 - Ergebnisse der Teilprobleme in Tabelleintragen
 - Zusammensetzen der Gesamtlösung



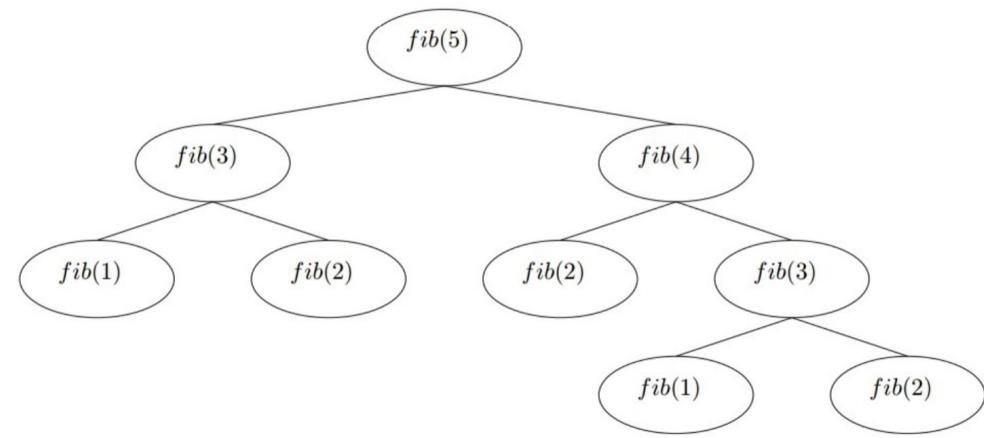
Fibonacci Folge

- Eine unendliche Folge von Zahlen
- Zahl durch Addition ihrer beiden Vorgänger ergibt
- Rekursive Beschreibung des Problemes:
 - $f_1 := 1$
 - $f_2 := 1$
 - $f_n := f_{n-1} + f_{n-2}$
- 1, 1, 2, 3, 5, 8, 13, ...



Fibonacci Folge

```
def fibonacci(n):
    """
        compute the fibonacci number
        n - a positive integer
        return the fibonacci number for a given n
    """
    #base case
    if n==0 or n==1:
        return 1
    #inductive step
    return fibonacci(n-1)+fibonacci(n-2)
```

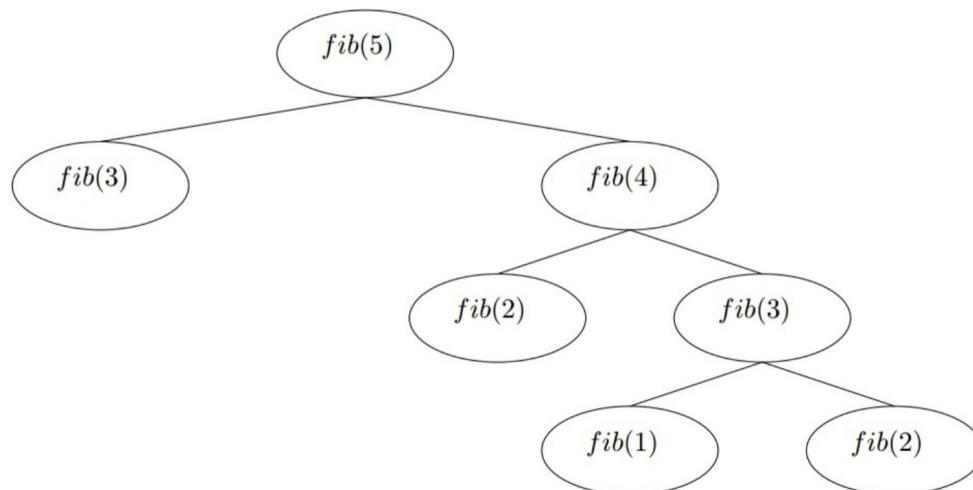


⇒ Exponentieller Aufwand $O(2^n)$



Dynamische Programmierung

- Rekursion
- Ergebnisse werden zwischengespeichert
 - Zuerst 'lookup' in Tabelle
- Laufzeit (**imac, i5**) für $\text{fib}(50) = 12586269025$
 - Standard: ~ 50m
 - DP: 0.00012s



⇒ Linearer Aufwand $O(n)$

```

import time

k = [0]*1000

def fib(n, k):
    if k[n] != 0: return k[n]
    if (n== 1 or n == 2): return 1
    k[n] = fib(n-1, k) + fib(n-2, k)
    return k[n]

def fib2(n):
    if (n == 1 or n == 2):
        return 1
    return fib2(n-1) + fib2(n-2)

start = time.time()
print(fib2(50))
end = time.time()
print("%.5f" % (end - start))

start = time.time()
print(fib(50,k))
end = time.time()
print("%.5f" % (end - start))
  
```