Vorkurs Informatik

Dirk Frettlöh

dfrettloeh@techfak.de

21. September 2023

Hangman
- Wichtige Fragen
- der Ablauf
- die Implementierung

Klassen
- erste Klasse erstellen

Hangman die Zweite
- Klasse
- Konstruktor
- read File
- print
- main

Vererbung
- Konzept
- Umsetzung

Anwendungen

Hangman (Galgenmännchen)

Nachdem wir das erste Spiel programmiert haben, starten wir gleich mit dem Nächsten.

Auch hier fangen wir wieder mit den Fragen an.

Auch hier fangen wir wieder mit den Fragen an.

Wichtige Fragen, die man sich vorher stellen sollte:

1. Welche Regeln hat das Spiel?
2. Wie sieht die Eingabe aus?
3. Wie sieht die Ausgabe aus?
4. Was müssen wir speichern?
5. Was ist der Ablauf?

1) Welche Regeln hat das Spiel?

1) Welche Regeln hat das Spiel?

- Das Programm wählt ein Wort aus.

1) Welche Regeln hat das Spiel?

- Das Programm wählt ein Wort aus.
- Der Benutzer rät entweder einen Buchstaben oder versucht, das Wort zu raten.

1) Welche Regeln hat das Spiel?

- Das Programm wählt ein Wort aus.
- Der Benutzer rät entweder einen Buchstaben oder versucht, das Wort zu raten.
- Wenn der Buchstabe nicht in dem Wort ist oder der Lösungsversuch falsch ist, wird der Rundenzähler reduziert.

1) Welche Regeln hat das Spiel?

- Das Programm wählt ein Wort aus.
- Der Benutzer rät entweder einen Buchstaben oder versucht, das Wort zu raten.
- Wenn der Buchstabe nicht in dem Wort ist oder der Lösungsversuch falsch ist, wird der Rundenzähler reduziert.
- Wenn es keinen Buchstaben mehr zu raten gibt oder der Lösungsversuch richtig ist, hat der Spieler gewonnen.

1) Welche Regeln hat das Spiel?

- Das Programm wählt ein Wort aus.
- Der Benutzer rät entweder einen Buchstaben oder versucht, das Wort zu raten.
- Wenn der Buchstabe nicht in dem Wort ist oder der Lösungsversuch falsch ist, wird der Rundenzähler reduziert.
- Wenn es keinen Buchstaben mehr zu raten gibt oder der Lösungsversuch richtig ist, hat der Spieler gewonnen.
- Fällt der Rundenzähler auf 0, hat der Spieler verloren.

2) Wie sieht die Eingabe aus?

2) Wie sieht die Eingabe aus?

Das Programm braucht eine Liste von Wörtern.
Der Benutzer muss in jeder Runde einen Buchstaben oder ein Lösungswort eingeben.

3) Wie sieht die Ausgabe aus?

3) Wie sieht die Ausgabe aus?

Die Ausgabe können wir mit *print()* realisieren, jedoch brauchen wir mehrere Ausgaben:

1. die Rundennummer und Infos zur Eingabe nach jedem Versuch des Spielers,

3) Wie sieht die Ausgabe aus?

Die Ausgabe können wir mit *print()* realisieren, jedoch brauchen wir mehrere Ausgaben:

1. die Rundennummer und Infos zur Eingabe nach jedem Versuch des Spielers,
2. " Gewonnen", wenn der Spieler das Wort richtig geraten hat,

3) Wie sieht die Ausgabe aus?

Die Ausgabe können wir mit *print()* realisieren, jedoch brauchen wir mehrere Ausgaben:

1. die Rundennummer und Infos zur Eingabe nach jedem Versuch des Spielers,
2. " Gewonnen", wenn der Spieler das Wort richtig geraten hat,
3. " Verloren", wenn der Spieler es nicht geschafft hat, in der vorgegebenen Rundenzahl das richtige Wort zu raten,

4) Was müssen wir speichern?

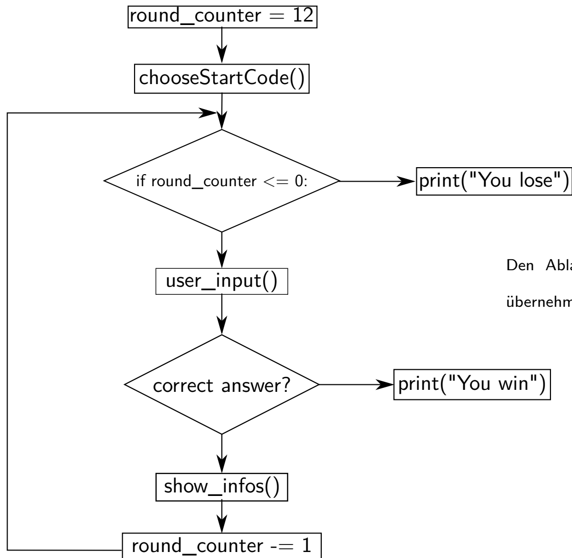
- das zu erratende Wort,

4) Was müssen wir speichern?

- das zu erratende Wort,
- die schon geratenen Buchstaben,

4) Was müssen wir speichern?

- das zu erratende Wort,
- die schon geratenen Buchstaben,
- das zu erratende Wort, bei dem die Buchstaben, die schon geraten wurden, angezeigt werden,



Den Ablauf können wir von Mastermind übernehmen.

Zuerst legen wir Variablen an, mit den zu speichernden Informationen.

```
1 getestet=[]
2
3 def spielen():
4     runden=10
5     info=[]
```

Dann müssen wir noch ein Wort wählen.

```
getestet=[]

def spielen():
    runden=10
    info=[]
    wort=wortwahl()
    wort=wort.lower()
    for i in range(len(wort)):
        info.append("_")
```

In dem `info` wird der aktuelle Informationsstand gespeichert, den der Spieler hat - also zu Beginn die Länge des Wortes.

Zuerst schreiben wir eine Funktion, die sich um die Ausgabe kümmert.

```
getestet=[]

def spielen():
    runden=10
    info=[]
    wort=wortwahl()
    wort=wort.lower()
    for i in range(len(wort)):
        info.append("_")

    while(runden>=0):
        zeigeinfo(runden,info)
```

```
1  def zeigeinfo(round_counter, word_to_play):
2      print("-" * 80)
3      print("Noch", round_counter, "Versuche", end="")
4      print("\t\t\tSchon␣versucht:", end="")
5      for i in getestet:
6          print(i + ",", end="")
7      print("\n\n\t", end="")
8      for i in word_to_play:
9          print(i + "␣", end="")
10     print("\n\n")
```

Steuersignale

Symbole	Signal für
`\n`	Zeilenumbruch
`\t`	Tabulator
`\b`	letztes Zeichen löschen
`\r`	Zum Anfang der Zeile gehen
`\v`	Vertikaler Tabulator

Als nächstes fügen wir eine Funktion hinzu,die auf Eingabe reagiert.

```
getestet=[]

def spielen():
    runden=10
    info=[]
    wort=wortwahl()
    wort=wort.lower()
    for i in range(len(wort)):
        info.append("_")

    while(runden>=0):
        zeigeinfo(runden,info)
        if (raten(wort,info)):
            runden-=1

        if(info.count("_")==0):
            break
    if (runden<=0):
        print("Verloren")
    else:
        print("Gewonnen in ",10-runden," Versuchen.")
```

```
1  def raten(word, info):
2      inp = input("Du bist dran:")
3      # Eingabe ueberpruefen
4      if (len(inp) != 1 and len(inp) != len(word)):
5          print("Bitte nur einen Buchstaben eingeben, oder ein moegliches 
               Loesungswort")
6          return False
7      inp=inp.lower()
8      # Der Spieler raet einen Buchstaben
9      if (len(inp) == 1):
10         # Der Buchstabe kommt nicht vor und ist noch nicht versucht worden:
11         if ((not inp in word) and (not inp in getestet)):
12             getestet.append(inp)
13             getestet.sort()
14             return True
15         else:
16             for i in range(0, len(word)):
17                 if (word[i] == inp):
18                     info[i] = word[i]
19             return True
20     else:    # Der Spieler raet das Loesungswort
21         if (word == inp):
22             for i in range(0, len(word)):
23                 info[i] = word[i]
24             return True
25         else:
26             getestet.append(inp)
27     return True
```

```
1 from random import randrange
2
3 def wortwahl():
4     f = open("hangman.txt")
5     words = f.readlines()
6     f.close()
7     return words[randrange(0, len(words))]
```

Informationen zu open

Mit dem Befehl *open* ist es möglich eine Datei zu öffnen, um aus ihr zu lesen oder in die sie zu schreiben.

Bei dem *open* Befehl kann man zusätzlich mit angeben, ob man die Datei nur lesend oder auch schreibend öffnen will.

```
1 open("hangman.txt", mode="r") # nur lesen (Wenn man nichts angibt)
2 open("hangman.txt", mode="w") # schreibend
3 open("hangman.txt", mode="a") # auch schreibend, aber wenn die Datei schon
4                                #existiert wird es am Ende der Datei angehaengt
```

Fertig?!

```python
def show_infos(round_counter, word_to_play):
```

Es ist nicht wirklich schön, dass die Variablen immer übergeben werden müssen.
Zudem ist der Quellcode nicht wirklich übersichtlich.

Fertig?!

```
1 def show_infos(round_counter, word_to_play):
```

Es ist nicht wirklich schön, dass die Variablen immer übergeben werden müssen.
Zudem ist der Quellcode nicht wirklich übersichtlich.
Um dieses Problem zu lösen, können wir unser Spiel in eine Klasse auslagern.

Was ist eine Klasse?

Was ist eine Klasse?

Wikipedia

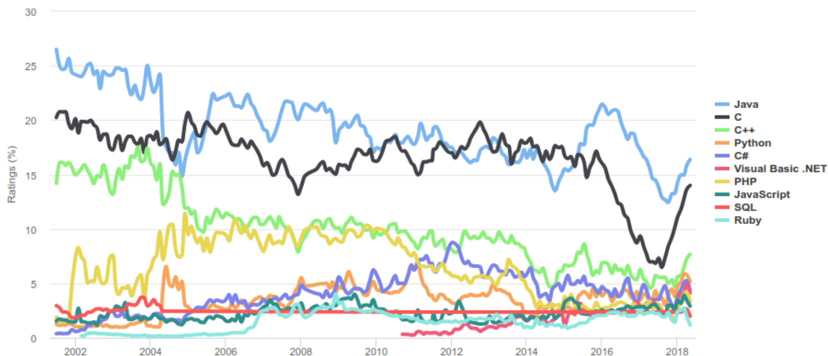
Unter einer Klasse (auch Objekttyp genannt) versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen Bauplan für eine Reihe von ähnlichen Objekten.

Objektorientierte Programmierung: ein *programming paradigm*.

Also eher eine Philosophie, oder ein Programmierstil.

Im Zentrum der Betrachtung stehen Objekte, nicht Funktionen.

Fast alle Sprachen können auch objektorientiert (java, C++,python....)



eine einfache Klasse in Python:

```python
class KlassenName():

    def test(self):
        print("ich bin eine Klasse")
```

eine einfache Klasse in Python:

```
1 class KlassenName():
2 
3     def test(self):
4         print("ich bin eine Klasse")
```

Das *class* signalisiert, dass es sich um eine Klasse handelt.
Der *KlassenName* kann durch einen frei gewählten Namen ausgetauscht werden. Jedoch sollte der Name mit einem großen Buchstaben beginnen.
Die Funktion/Methode in Zeile 3 kennen wir schon. Das Einzige, dass sich dort geändert hat, ist der Parameter *self*.

Wie benutzt man jetzt diese Klasse?

```
my_class = KlassenNamen()
my_class.test()
```

Wie benutzt man jetzt diese Klasse?

```
1 my_class = KlassenNamen()
2 my_class.test()
```

Bis jetzt fragt ihr euch zurecht, warum man das machen sollte. Um das klären zu können, benötigen wir ein etwas umfangreicheres Beispiel.

Beispiel

```
1  class  Beispiel():
2      def __init__(self,text):
3          self.text = text
4
5      def ausgabe(self):
6          print("Ich habe folgenden Inhalt", self.text)
7
8  aBeispiel = Beispiel("Klasse A")
9  bBeispiel = Beispiel("Klasse B")
10 aBeispiel.ausgabe()
11 bBeispiel.ausgabe()
```

Was zeigt uns dieses Beispiel?

Beispiel

```
1  class  Beispiel():
2      def __init__(self,text):
3          self.text = text
4
5      def ausgabe(self):
6          print("Ich habe folgenden Inhalt", self.text)
7
8  aBeispiel = Beispiel("Klasse A")
9  bBeispiel = Beispiel("Klasse B")
10 aBeispiel.ausgabe()
11 bBeispiel.ausgabe()
```

Was zeigt uns dieses Beispiel?

- In einer Klasse können wir Daten speichern.
- Von einer Klasse können wir mehrere *Instanzen* erzeugen.
- In jeder dieser *Instanzen* können unterschiedliche Daten gespeichert werden.

Gehen wir schrittweise vor.

```
class Beispiel()
```

1) Wir definieren eine Klasse mit dem Namen *Beispiel*.

Gehen wir schrittweise vor.

```
1 class Beispiel()
2     def __init__(self,text):
```

2) Mit der Funktion `__init__` , die keine Funktion ist, sondern ein Konstruktor, können wir angeben, welche Parameter unsere Klasse bekommen muss. (Das *self* muss da stehen)

Gehen wir schrittweise vor.

```
1 class Beispiel()
2     def __init__(self,text):
3         self.text = text
```

3) Innerhalb des Konstruktors werden die Variablen angelegt. Wenn man mit Klassen arbeitet, spricht man häufig von Attributen. Das sind Variablen, die in einer Klasse gespeichert werden.

Gehen wir schrittweise vor.

```
1 class Beispiel()
2     def __init__(self,text):
3         self.text = text
4 
5     def ausgabe(self):
6         print("Ich habe folgenden Text bekommen", self.text)
```

4) Innerhalb der Klassen können Funktionen angelegt werden, die mit den Variablen (Attributen) innerhalb der Klasse rechnen und/oder von außen Informationen bekommen.
(Wenn eine Funktion in einer Klasse ist, wird sie als Methode bezeichnet.)

das *self*

Wenn man eine Klasse erstellt, ist es zwingend nötig, dass der erste Parameter jeder Methode *self* ist. Das *self* ist eigentlich nur ein Zeiger auf sich selbst.

Ein ausführlicheres Beispiel:

Python kennt ganze Zahlen (`int`) und reelle Zahlen (`float`), aber auch komplexe Zahlen?

Strenggenommen kennt jeder Computer die meisten reellen Zahlen nicht, aber dazu mehr im ersten Semester.

Außerdem gibt es für python natürlich Bibliotheken mit komplexen Zahlen, aber wir tun mal so, als ob nicht.

Wir erstellen also eine Klasse dafür.

```
1 class Komplex():
2 
3     def __init__(self,real,imag):
```

class Komplex

Was soll ein Element dieser Klasse sinnvollerweise können?
Vielleicht Ihren Real- und Imaginärteil kennen:

```
1  class Komplex():
2
3      def __init__(self,real,imag):
4          self.real = real
5          self.imag = imag
6
7  #########################
8
9  z=Komplex(2.5,7)
10 print(z.real)
```

```
2.5
```

class Komplex

Wir wollen ja auch die Zahl selbst ausgeben.

```
class Komplex():

    def __init__(self,real,imag):
        self.real = real
        self.imag = imag

###########################

z=Komplex(2.5,7)
print(z)
```

```
<__main__Komplex object at 0x10bcb0820>
```

Ups.

class Komplex

Problem: `print` braucht den Typ string oder int oder float. Bekommt aber Komplex, und weiß nicht, was tun. Also müssen wir eine eigene Ausgabe für die Klasse Komplex definieren, oder aber auf Anfrage einen string zurückgeben. Wir tun mal beides:

```
class Komplex():

    def __init__(self,real,imag):
        self.real = real
        self.imag = imag
    def show(self):
        print(str(self.real)+'+'+str(self.imag)+'i')
    def __str__(self):
        return str(self.real)+'+'+str(self.imag)+'i'

############################

z=Komplex(2.5,7)
z.show()
print(z)
```

```
2.5+7i
2.5+7i
```

class Komplex

```
1  class Komplex():
2
3      def __init__(self,real,imag):
4          self.real = real
5          self.imag = imag
6      def show(self):
7          print(str(self.real)+'+'+str(self.imag)+'i')
8      def __str__(self):
9          return str(self.real)+'+'+str(self.imag)+'i'
10
11 ###########################
12
13 z=Komplex(2.5,7)
14 z.show()
15 print(z)
```

Wir sehen im Moment zwei Attribute (`real, imag`), und eine Methode (show)

In vielen objektorientierten Programmiersprachen greift man auf die Attribute so zu:
`z.real`

und auf die Methoden so: `z.show()`

class Komplex

Wir wollen nun natürlich auch eine komplexe Zahl komplex konjugieren, sowie zwei komplexe Zahlen addieren oder multiplizieren.

```
1  class Komplex():
2  
3      def __init__(self,real,imag):
4          self.real = real
5          self.imag = imag
6  
7      def add(self,other):
8          return Komplex(self.real+other.real,self.imag+other.imag)
9      def mult(self,other):
10         return  Komplex(self.real*other.real-self.imag*other.imag,self.imag*
               other.imag+self.imag*other.real)
11     def conj(self):
12         return Komplex(self.real,-self.imag)
13 
14  [...]
15 
16 ##########################
17 
18 z=Komplex(2.5,7)
19 y=Komplex(3,-2)
20 x=z.conj()
21 x.show()
22 x=z.add(y)
23 x.show()
24 x=z.mult(y)
25 x.show()
```

class Komplex

```
1  class Komplex():
2
3      def __init__(self,real,imag):
4          self.real = real
5          self.imag = imag
6
7      def add(self,other):
8          return Komplex(self.real+other.real,self.imag+other.imag)
9      def mult(self,other):
10         return  Komplex(self.real*other.real-self.imag*other.imag,self.imag*
               other.imag+self.imag*other.real)
11     def conj(self):
12         return Komplex(self.real,-self.imag)
13
14 [...]
15
16 ############################
17
18 z=Komplex(2.5,7)
19 y=Komplex(3,-2)
20 x=z.conj()
21 x.show()
22 x=z.add(y)
23 x.show()
24 x=z.mult(y)
25 x.show()
```

```
2.5+-7i
5.5+5i
21.5+7i
```

class Komplex

Das `2.5+-7i` geht schöner:

```
class Komplex():

    def __init__(self,real,imag):
        self.real = real
        self.imag = imag

[...]

    def show(self):
        if self.imag < 0:
            print(str(self.real)+str(self.imag)+"i")
        else:
            print(str(self.real)+"+"+str(self.imag)+"i")

    def __str__(self):
        return str(self.real)+'+'+str(self.imag)+'i'

##############################

z=Komplex(2.5,7)
y=Komplex(3,-2)
x=z.conj()
x.show()
```

```
2.5-7i
5.5+5i
21.5+7i
```

Könnt ihr euch noch an diese Zeile erinnern?

```
1 def show_infos(round_counter , word_to_play):
```

Wir bauen jetzt aus unserem Hangman-Game eine Klasse.

```
1 def show_infos(self,round_counter , word_to_play):
```

1) der Klassen Kopf

```
1 from random import randrange
2
3 class Hangman():
```

2) der Konstruktor

```
1    def __init__(self,rundenanz):
2        self.getestet=[]
3        self.runden=rundenanz
4        self.info=[]
```

Alle Variablen die wir benötigen, werden hier angelegt.

3) die wortwahl-Funktion jetzt als Methode

```
1     def wortwahl(self):
2         f = open("hangman.txt")
3         words = f.readlines()
4         f.close()
5         return words[randrange(0, len(words))]
```

4) die zeigeinfo-Methode

```
1      def zeigeinfo(self):
2          print("-" * 80)
3          print("Noch", self.runden, "Versuche", end="")
4          print("\t\t\tSchon␣versucht:", end="")
5          for i in self.getestet:
6              print(i + ",", end="")
7          print("\n\n\t", end="")
8          for i in self.info:
9              print(i + "␣", end="")
10         print("\n\n")
```

5) die nextRound Methode

```
1      def raten(self):
2         inp = input("Du␣bist␣dran:")
3         # Eingabe ueberpruefen
4         if (len(inp) != 1 and len(inp) != len(self.wort)):
5             print("Bitte␣nur␣einen␣Buchstaben␣eingeben,␣oder␣ein␣moegliches␣
                  Loesungswort")
6             return False
7         inp=inp.lower()
8         # Der Spieler raet einen Buchstaben
9         if (len(inp) == 1):
10            # Der Buchstabe kommt nicht vor und ist noch nicht versucht worden:
11            if ((not inp in self.wort) and (not inp in self.getestet)):
12                self.getestet.append(inp)
13                self.getestet.sort()
14                return True
15            else:
16                for i in range(0, len(self.wort)):
17                    if (self.wort[i] == inp):
18                        self.info[i] = self.wort[i]
19                return True
20        else:    # Der Spieler raet das Loesungswort
21            if (self.wort == inp):
22                for i in range(0, len(self.wort)):
23                    self.info[i] = self.wort[i]
24                return True
25            else:
26                self.getestet.append(inp)
27        return True
```

6) unsere main als Methode

```
1       def spielen(self):
2           self.wort=self.wortwahl()
3           self.wort=self.wort.lower()
4           for i in range(len(self.wort)):
5               self.info.append("_")
6           while(self.runden>=0):
7               self.zeigeinfo()
8               if (self.raten()):
9                   self.runden-=1
10
11              if(self.info.count("_")==0):
12                  break
13          if (self.runden<=0):
14              print("Verloren")
15          else:
16              print("Gewonnen in ",10-self.runden," Versuchen.")
```

7) So startet man jetzt eine Runde.

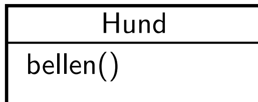
```
myHangman = Hangman(10)
myHangman.spielen()
```

Was hat sich jetzt geändert?

- Alle Funktionen wurden eine Ebene weiter eingerückt.
- Die Funktionsparameter wurden auf ein *self* reduziert.
- Innerhalb der Funktionen wurde jeder Variable ein *self.* vorangestellt.

Sonst hat sich an unserem Quellcode nichts geändert.

Vererbung

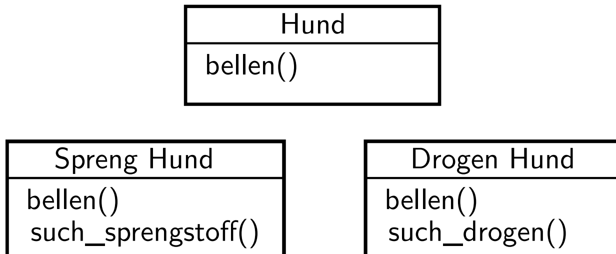


Stellen wir uns vor: Wir haben eine Klasse Hund. Dieser Hund kann bellen.

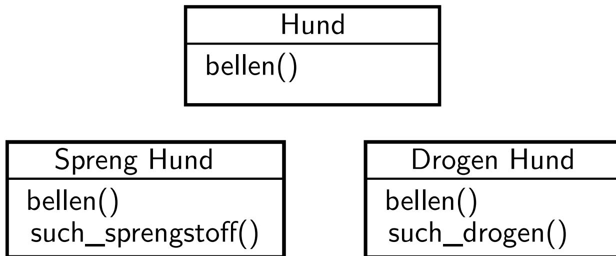
Hund
bellen()

Drogen Hund
bellen() such_drogen()

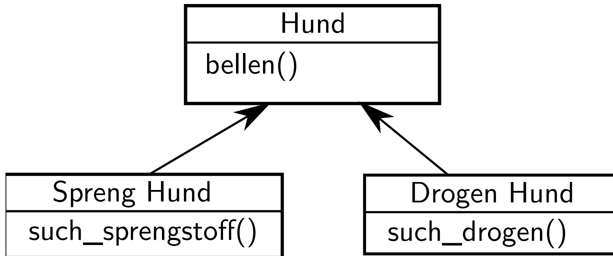
Jetzt benötigen wir noch einen Drogenhund und dieser kann natürlich auch bellen, aber zusätzlich noch Drogen suchen.



Zum Schluss bauen wir uns noch einen Sprenghund, der auch bellen kann. Zusätzlich kann er Sprengstoffe aufspüren.



Wenn wir für jeden Hund eine Klasse schreiben würden, müssten wir die *bellen* Methode dreimal schreiben. Wie ihr euch denken könnt, ist das nicht sinnvoll.



Besser wäre es, wenn unser Sprenghund und unser Drogenhund von unserem Hund das Bellen "erben" könnten.

Unsere Hund-Klasse sieht folgendermaßen aus:

```
1 class Hund():
2
3     def __init__(self, name):
4         self.name = name
5
6     def bellen(self):
7         print("Wau Wau")
```

und so unser Drogenhund:

```
1 class DrogenHund(Hund):
2
3     def __init__(self, name):
4         self.name = name
5
6     def such_drogen(self):
7         print(self.name, "sucht nach Drogen")
```

zum Schluss noch unser Sprenghund:

```python
class SprengHund(Hund):

    def __init__(self, name):
        self.name = name

    def such_sprengstoffe(self):
        print(self.name, "sucht nach Sprengstoffen")
```

zum ausprobieren benutzen wir:

```python
hund1 = Hund("waldi")
hund2 = DrogenHund("bello")
hund3 = SprengHund("bruno")

hund1.bellen()
hund2.bellen()
hund3.bellen()

hund2.such_drogen()
hund3.such_sprengstoffe()
```

Wenn man sich jetzt überlegt, dass die Hunde eher "wuff wuff" machen als "wau wau", dann muss man nur an einer Stelle etwas ändern.

```
1 class Hund():
2
3     def __init__(self, name):
4         self.name = name
5
6     def bellen(self):
7         print("Wuff wuff")
```

Anwendungen

Zum Schluss: wie benutzen wir python im Alltag?

Anwendungen

Zum Schluss: wie benutzen wir python im Alltag?

(Nicht von mir, von einem echten Tüftler:)

Lautsprecher mit Akku und Speicherkarte. Spielt Lieder in alphabetischer Reihenfolge.

Keine Zufallsreihenfolge ("shuffle" oder "random shuffle") vorgesehen.

Anwendungen

Zum Schluss: wie benutzen wir python im Alltag?

(Nicht von mir, von einem echten Tüftler:)

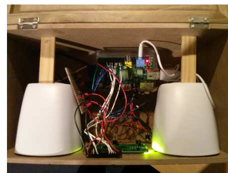
Lautsprecher mit Akku und Speicherkarte. Spielt Lieder in alphabetischer Reihenfolge.

Keine Zufallsreihenfolge ("shuffle" oder "random shuffle") vorgesehen.

Software ist python! Also 40 Zeilen Code ergänzen, Lieder zufällig umbennen (import random).

Anwendungen

Derselbe Tüftler:



“Aus einem Raspberry Pi, einem LCD Display, einem Wlan-stick, PC Lautsprechern und 4 Knöpfen und 140 Zeilen Python-Code habe ich ein Web-Radio gebaut.”

Anwendung Vorlesung

Ich selbst bin mehr so theoretisch orientiert. Meine Anwendungen:

Vorlesung Kryptographie: ein interaktives "sagemath notebook" benutzt als Sprache python, und verbindet das mit mächtigen Mathewerkzeugen (R für Statistik, numpy, scipy: python-Pakete für Mathe, GAP, Maxima für Algebra...)

Anwendung Vorlesung

Ich selbst bin mehr so theoretisch orientiert. Meine Anwendungen:

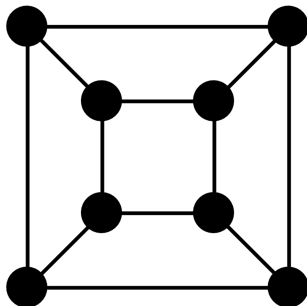
Vorlesung Kryptographie: ein interaktives “sagemath notebook” benutzt als Sprache python, und verbindet das mit mächtigen Mathewerkzeugen (R für Statistik, numpy, scipy: python-Pakete für Mathe, GAP, Maxima für Algebra...)

Abgaben zu den Aufgabe waren erlaubt handschriftlich, LaTeX, python oder sagemath.

[Zeigen: `https://www.math.uni-bielefeld.de/~frettloe/teach/krypto/loes20-1.pdf`]

Anwendung Forschung

Forschung: Ein “Graph” ist ein Objekt mit Knoten und Kanten.



Anwendung Forschung

"Perfect colouring" eines Graphen: jeder weiße Knoten hat gleich viele weiße, rote und schwarze Nachbarn wie jeder andere weiße Knoten. Dito für schwarze und rote.

Anwendung Forschung

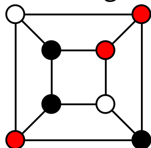
"Perfect colouring" eines Graphen: jeder weiße Knoten hat gleich viele weiße, rote und schwarze Nachbarn wie jeder andere weiße Knoten. Dito für schwarze und rote.

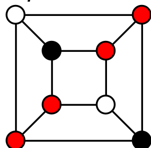
Links kein *perfect colouring*, rechts ein *perfect colouring*.

○ colour 1

● colour 2

● colour 3





$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 2 & 2 & 1 \end{pmatrix}$$

Anwendung Forschung

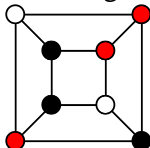
"Perfect colouring" eines Graphen: jeder weiße Knoten hat gleich viele weiße, rote und schwarze Nachbarn wie jeder andere weiße Knoten. Dito für schwarze und rote.

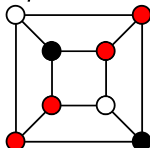
Links kein *perfect colouring*, rechts ein *perfect colouring*.

○ colour 1

● colour 2

● colour 3





$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 2 & 2 & 1 \end{pmatrix}$$

In einer Arbeit mit Joseph Damasco (Manila) bestimmten wir alle möglichen Farbkombinationen.

Dazu benutzten wir ausgiebig sagemath.

[Zeigen: `https://www.math.uni-bielefeld.de/~frettloe/papers/perf-gr-col.pdf`]

Fragen ?

Fragen?

Geschafft

Soviel zu Grundlagen von Python.

Viel Spaß damit!