

Objektorientierung und Funktionale Programmierung

Einführung

Kristof Van Laerhoven
kvl@eti.uni-siegen.de

Michael Moeller
Michael.moeller@uni-siegen.de

Andreas Hoffmann
Andreas.Hoffmann@uni-siegen.de
uni-siegen.de



Jetzt mit mehr Zeit: for-Schleifen

```
for i in range(5):  
    print(i)
```

```
0  
1  
2  
3  
4
```

Im Allgemeinen hat **range** einen Startpunkt, einen Endpunkt, und eine Schrittweite

```
for i in range(1,10,3):  
    print(i)
```

```
1  
4  
7
```

Python können **for**-Schleifen aber auch noch allgemeiner, nämlich für beliebige **iterierbare Objekte** verwendet werden. Unter den uns bekannten Objekten, sind dies z.B. Listen:

```
meine_liste = ['hallo', 4, [1,2,3]]  
for element in meine_liste:  
    print(element)
```

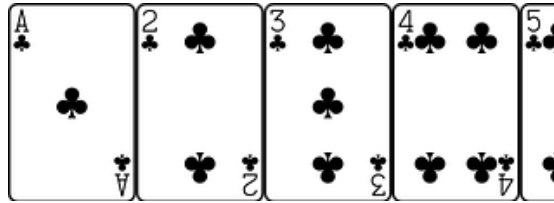
```
hallo  
4  
[1, 2, 3]
```

Mir **enumerate** kann zusätzlich auf den Index zugegriffen werden:

```
meine_liste = ['hallo', 4, [1,2,3]]  
for index, element in enumerate(meine_liste):  
    print('An index {} steht {}'.format(index, element))
```

```
An index 0 steht hallo  
An index 1 steht 4  
An index 2 steht [1, 2, 3]
```

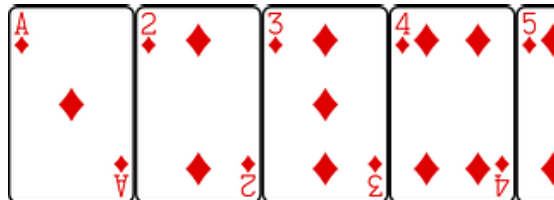
Spieler 1



Preiskarten



Spieler 2



Beim letzten Mal: Geschafft dieses Programm sinnvoll zum Laufen zu bringen. Aber ist dies *gut* programmiert?



```
1 from random import shuffle
2 kartenSpieler1 = [1,2,3,4,5,6,7,8,9,10,11,12,13] # das hier werden Listen an Integers;
3 langfristig ersetzen durch list(1,range(n+1)), mit n=13
4 #kartenSpieler2 = [1,2,3,4,5,6,7,8,9,10,11,12,13] # no need: KI-2 always play the price
5 card as its choice
6 preiskarten = [1,2,3,4,5,6,7,8,9,10,11,12,13]
7 shuffle(preiskarten)
8
9 ### "KI"-1: Spiele eine zufällige Karte:
10 shuffle(kartenSpieler1)
11
12 punkteSpieler1 = 0
13 punkteSpieler2 = 0
14
15 while (len(preiskarten)>0): # man könnte auch nur while (preiskarten): schreiben.
16
17     neuePreiskarte = preiskarten.pop()#Neue Preiskarte zufällig ziehen
18     neueKarteSpieler1 =kartenSpieler1.pop()
19     neueKarteSpieler2 = neuePreiskarte ### Vorsicht: hier müsste man aufpassen wenn sich
20 andere Teile des Programms ändern, z.B. andere Preiskarten als Spielkarten.
21
22     if (neueKarteSpieler1 > neueKarteSpieler2):
23         punkteSpieler1 = punkteSpieler1 + neuePreiskarte
24         print("Spieler 1 gewinnt die Runde. Neuer Punktestand: ", punkteSpieler1, "vs.",
25 punkteSpieler2)
26     elif (neueKarteSpieler2 > neueKarteSpieler1):
27         punkteSpieler2 = punkteSpieler2 + neuePreiskarte
28         print("Spieler 2 gewinnt die Runde. Neuer Punktestand: ", punkteSpieler1, "vs.",
29 punkteSpieler2)
30     else:
31         print("Gleichstand - nichts passiert! Neuer Punktestand: ", punkteSpieler1, "vs.",
32 punkteSpieler2)
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

Nächster Schritt

Existierenden Code etwas besser machen.

Mit größer werdenden Programmen wird es extrem schwierig Code zu verstehen und zu pflegen, wenn alles in einer Datei sequenziell hintereinander geschrieben ist. Ein deutlich besserer Stil ist das aufteilen in Sinnabschnitte mittels **Funktionen**.

```
def name_der_function(beliebig, viele, argumente):
```

```
    """
```

```
    Dies ist ein docstring. Hier kann dokumentiert werden was die Funktion tut
```

```
    """
```

```
    aller eingerückter Code wird ausgeführt
```

```
    return rueckgabewert (falls gewünscht)
```

```
def meine_funktion():  
    print('hallo')
```

```
meine_funktion()
```

```
In [1]: runfile('temp.py')  
hallo
```

```
def meine_funktion(name):  
    print('hallo', name)  
  
meine_funktion('zusammen!')
```

```
In [3]: runfile('temp.py')  
hallo zusammen!
```

Wir können in unserem Goofspiel einige Teile als Funktionen auslagern! Beispiele hierfür im Code!

```
def strategie_spieler_1(wert_der_preiskarte):  
    wert_karte_1 = karten_spieler_1.pop()  
    print('Spieler 1 spielt eine ', wert_karte_1)  
    return wert_karte_1  
  
def strategie_spieler_2(wert_der_preiskarte):  
    wert_karte_2 = wert_der_preiskarte  
    karten_spieler_2.remove(wert_karte_2)  
    print('Spieler 2 spielt eine ', wert_karte_2)  
    return wert_karte_2  
  
while (preiskarten):  
    wert_der_preiskarte = preiskarten.pop()  
    print('\nDer Wert der Preiskarte ist ', wert_der_preiskarte)  
  
    wert_karte_1 = strategie_spieler_1(wert_der_preiskarte)  
  
    wert_karte_2 = strategie_spieler_2(wert_der_preiskarte)
```

Ein solcher Code ist schon eine signifikante Verbesserung: Wir haben die Strategien der Spieler in eigenen Funktionen isoliert. Ein Programmierer kann nun die Strategie ändern ohne dass Hauptprogramm vollständig durcharbeiten oder zu verstehen! Um solche modularen Änderungen einfacher möglich zu machen, sollte die Funktionsweise von Funktionen mit dokumentiert werden. Dies geschieht über den **docstring**.

```
def auswertung_am_spielende(punkte_spieler_1,punkte_spieler_2):  
    """  
    Parameters  
    -----  
    punkte_spieler_1 : Anzahl der Punkte, die Spieler 1  
                      insgesamt erreicht hat.  
    punkte_spieler_2 : Anzahl der Punkte, die Spieler 1  
                      insgesamt erreicht hat.  
  
    Vergleicht die Anzahl der Punkte und gibt eine Nachricht über den  
    Gewinner oder ein Unentschieden auf der Konsole aus.  
  
    Returns  
    -----  
    None.  
    """  
  
    if punkte_spieler_1 > punkte_spieler_2:  
        print("Spieler 1 hat gewonnen!")  
    elif punkte_spieler_1 < punkte_spieler_2:  
        print("Spieler 2 hat gewonnen!")  
    else:  
        print('Dieses Spiel war unentschieden!')
```

```
In [20]: help(auswertung_am_spielende)
Help on function auswertung_am_spielende in module __main__:

auswertung_am_spielende(punkte_spieler_1, punkte_spieler_2)
    Parameters
    -----
    punkte_spieler_1 : Anzahl der Punkte, die Spieler 1
                      insgesamt erreicht hat.
    punkte_spieler_2 : Anzahl der Punkte, die Spieler 1
                      insgesamt erreicht hat.

    Vergleicht die Anzahl der Punkte und gibt eine Nachricht über den
    Gewinner oder ein Unentschieden auf der Konsole aus.

    Returns
    -----
    None.
```

Für längere Funktionen (und jeden anderen Code) macht es Sinn über den docstring hinaus noch **Kommentare** einzufügen. Dies geschieht in Python durch das Symbol #:

alles was hier steht wird vom Interpreter ignoriert

Dabei sollte darauf geachtet werden, dass sinnvolle (die Logik des Programms erklärende) Kommentare genutzt werden

```
# der Wert der Variablen b wird auf a gesetzt  
b = a
```

Hier wurde der Code zwar kommentiert, aber ohne dass der Kommentar irgendeinen Mehrwert liefert!

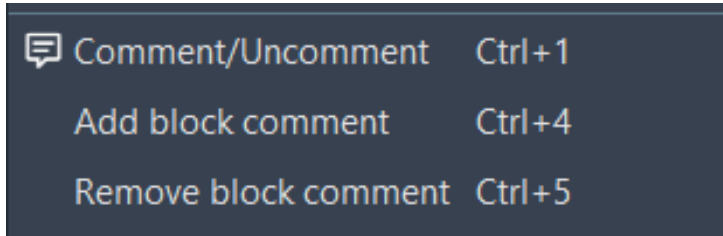
```
# der Variablen wert_der_preiskarte wird der Rückgabewert  
# der auf preiskarten aufgerufenen Funktion pop zugeordnet  
wert_der_preiskarte = preiskarten.pop()  
  
# die letzte der Preiskarten wird als wert_der_preiskarte  
# verwendet und aus der Liste preiskarten gelöscht.  
wert_der_preiskarte = preiskarten.pop()
```

schlecht

besser

Die meisten Editoren und IDEs haben eine Funktionalität alle gerade markierten Zeilen aus- bzw. einzukommentieren.

Z.B. Spyder unter Windows:



Alternativ kann man die Notation des Docstrings (drei Anführungszeichen) auch für Blockkommentare nutzen. Formal ist dies in Python ein String über mehrere Zeilen, der einfach keiner Variablen zugeordnet wird.

```
"""  
Hier wird irgendetwas kommentiert  
a = 1  
b = 2  
c = 3  
"""
```

positional arguments vs. keyword arguments

In Python kann man Funktionen (wie vorhin gesehen) mit einer Reihe von Eingabeargumenten versehen.

Beim Aufruf muss dann die Reihenfolge der Eingabeargumente beibehalten werden

– man spricht von *positional arguments*.

```
def meine_funktion(grußwort, name):  
    print(grußwort, name)
```

```
meine_funktion('Hallo', 'zusammen')  
meine_funktion('zusammen', 'Hallo')
```

```
Hallo zusammen  
zusammen Hallo
```

Man kann aber den in der Funktion definierten Eingabeargumenten auch default (initial-)Werte geben. In diesem Fall sind sie unter dem in der Definition genannten Namen bekannt, also sogenannte *keyword arguments*, und können mittels ihres Namens in beliebiger Reihenfolge mit Werten versehen werden.

```
def meine_funktion(grußwort='Hallo', name=''):  
    print(grußwort, name)
```

```
meine_funktion()  
meine_funktion(name = 'zusammen')  
meine_funktion(grußwort='Hi')  
meine_funktion(grußwort='Hi', name='zusammen')
```

```
Hallo  
Hallo zusammen  
Hi  
Hi zusammen
```

Für beide, positional arguments und keyword arguments, kann man mittels * bzw. ** eine beliebige Anzahl von Eingabeargumenten erlauben!

`def name_der_function(*args):`

Hier steht eine Funktion, die mit beliebig vielen in args gespeicherten Positionalen Argumenten umgehen kann.

`def name_der_function(**kwargs):`

Hier steht eine Funktion, die mit beliebig vielen in kwargs gespeicherten keyword Argumenten umgehen kann.

```
def meine_funktion(*args):  
    print(type(args))  
    for argument in args:  
        print(argument)  
meine_funktion('hallo', 4*4, 'zusammen!', [3,2,1])
```

```
def meine_funktion(**kwargs):  
    print(type(kwargs))  
    for key,value in kwargs.items():  
        print('unter dem namen "{}" steht der wert "{}"'.format(key, value))
```

Man kann explizit angeben welche Eingabe und Ausgabetypen eine Funktion bekommt und liefert

```
def name_der_function(variable1: typ1, variable2: typ2) -> ausgabeTyp:
```

Dies geht übrigens auch für Variablen

```
a: int = 3  
name: str = "Michael"  
meine_liste: list = [1, 'hallo', 5.342]
```

Dies sind Hinweise, die auf den Ablauf des Programms KEINEN Einfluss haben. Insbesondere führt

```
a: int = 3.1415
```

(oder ein späteres Überschreiben eines `a: int`) nicht zu einem Fehler. Es gibt jedoch Möglichkeiten/Module, die dies können/zum Debuggen verwenden!

<https://docs.python.org/3.14/library/typing.html#module-typing> für Details!

Eine interessante Eigenschaft von Python ist, dass Funktionen auch als Funktionsobjekte in Variablen gespeichert werden können!

```
def say_hello(name):  
    print('Hello', name)  
  
meine_funktion = say_hello
```

```
print(meine_funktion)
```

Hexadezimale Speicheradresse der Funktion

```
<function say_hello at 0x0000016B54407C10>
```

```
meine_funktion('Michael')
```

```
Hello Michael
```

meine_funktion kann nun genauso genutzt werden wie say_hello

```
print(meine_funktion('Michael'))
```

```
Hello Michael  
None
```

Da die Funktion say_hello (und damit auch meine_funktion) keinen Rückgabewert hat, wird hier None auf der Konsole ausgegeben.

Funktionen als Variablen zu speichern ermöglicht die Konstruktion von **Funktionen höherer Ordnung (higher order function)**, d.h. Funktionen die als ein Eingabeargument eine andere Funktion nehmen.

Beispiel: Schreibe Funktionen zum Addieren und Multiplizieren von Zahlen, sowie eine Funktion „anwenden“ die eine gegebene Funktion auf zwei Zahlen anwendet.

```
def addiere(x,y):  
    return x+y  
  
def multipliziere(x,y):  
    return x*y  
  
def anwenden(x,y,funktion):  
    return funktion(x,y)  
  
meine_funktion = multipliziere  
print(anwenden(2,3,meine_funktion))
```

Idee: Die in einer Variablen gespeicherten Funktion könnte sich innerhalb des Programms (z.B. durch Benutzereingaben) ändern.

Unser vorheriger Goofspiel-Code hat die Strategien der Spieler (hier letzte Karte spielen bzw. Wert der Preiskarte spielen) immer noch hart festgelegt (siehe unten). Wie könnte man dies (insbesondere bei wachsender Anzahl an KI-Strategien oder menschlichen Inputs) verbessern?

```
def strategie_spieler_1(wert_der_preiskarte):  
    wert_karte_1 = karten_spieler_1.pop()  
    print('Spieler 1 spielt eine ', wert_karte_1)  
    return wert_karte_1  
  
def strategie_spieler_2(wert_der_preiskarte):  
    wert_karte_2 = wert_der_preiskarte  
    karten_spieler_2.remove(wert_karte_2)  
    print('Spieler 2 spielt eine ', wert_karte_2)  
    return wert_karte_2  
  
while (preiskarten):  
    wert_der_preiskarte = preiskarten.pop()  
    print('\nDer Wert der Preiskarte ist ', wert_der_preiskarte)  
  
    wert_karte_1 = strategie_spieler_1(wert_der_preiskarte)  
  
    wert_karte_2 = strategie_spieler_2(wert_der_preiskarte)
```

Mit wachsender Programmgröße ist es oft sinnvoll bestimmte Funktionen in einer Datei (in der nicht das Hauptprogramm steht) zusammenzufassen.

Um diese in einem anderen Programm verwenden zu können, benutzt man den Befehl `import`

Wie schon zuvor bei `import random` gesehen, wird der Name der importierten Datei dann zum **Namespace** für alle darin enthaltenen Funktionen (erinnern sie sich an `random.shuffle(karten)`).

Mit `from (name_des_moduls) import (name_der_funktion)` kann man Funktionen verwenden ohne dass diese in einem Namespace sind.

Mit `import (name_des_moduls) as (name_des_gewünschten_namespaces)` kann man den Namespace der im importierten Modul enthaltenen Funktionen selbst bestimmen (und vom Modulnamen abweichen).

```
import random Datei strategien.py

def spiele_wert_der_preiskarte(karten, wert_der_preiskarte):
    wert_karte = wert_der_preiskarte
    karten.remove(wert_karte)
    return wert_karte

def spiele_zufaellige_karte(karten, wert_der_preiskarte):
    wert_karte = random.sample(karten, 1)[0]
    karten.remove(wert_karte)
    return wert_karte
```

Im Hauptprogramm:

```
import strategien

strategie_spieler_1 = strategien.spiele_zufaellige_karte
strategie_spieler_2 = strategien.spiele_wert_der_preiskarte
```

Beim **import** Befehl muss die Datei im gleichen Ordner wie das ausgeführte Skript sein (oder im PYTHONPATH sein). Für Unterordner kann man (Name des Unterordners) • (Name der Datei) verwenden.

Zum Beispiel

```
import subdirectory.strategien as strat

strategie_spieler_1 = strat.spiele_zufaellige_karte
strategie_spieler_2 = strat.spiele_wert_der_preiskarte
```

`for` Schleifen über iterierbare Objekte, z.B. Listen

`enumerate` für Schleifen über Indizes und Werte

Funktionsdefinitionen `def` `return`

Docstrings, `help`

Kommentare `#` `"""` `"""`

Positional und Key-Word Argumente, beliebige Anzahl mit `*args` `**kwargs`

Type hints `variable1: typ1`

Funktionen in Variablen speichern

Funktionen die Funktionen entgegennehmen oder zurückgeben (higher order functions)

Importieren von Dateien/Modulen, `import`, Unterordner mit `.`

```
from (name_des_moduls) import (name_der_funktion)
```

```
import (name_des_moduls) as (name_des_gewünschten_namespaces)
```