

1 Python und IDLE

Höhere Programmiersprache Python

IDLE – Integrated Development and Learning Environment

Shell zum Testen von Programmzeilen

ALT + p: Letzte Programmzeile wiederholen

ALT + n: Eine Programmzeile vorwärts

STRG + F6: Restart Shell

File/New oder File/Open ruft den Editor auf: Programmzeilen auf dem Rechner speichern

F5 führt das Programm aus

Zeichenketten (Strings):

"dies ist eine Zeichenkette"

'dies ist auch eine Zeichenkette'

Bitte entweder ' oder " verwenden – nicht mischen!

Variablen

vorname = "Martin"

print(vorname)

Backslash

\n bedeutet Zeilenvorschub

Den Backslash erhalte ich durch \\\

2 Unterschied zwischen Funktionen und Methoden

Funktionen

Die Funktionen print(), len() usw. sind in Python fest eingebaut

type(vorname) zeigt den Typ der Variablen an

Methoden

Objekte (z. B. str) besitzen Funktionen, die mit den Objektdaten arbeiten – Methoden

Beispiel: vorname.lower() Denkt an die Klammern!

help(str) zeigt alle Methoden der Klasse str, dir(str) zeigt eine kürzere Übersicht

3 Rechnen mit Zeichenketten und Zahlen

Rechnen mit Zeichenketten

Die Operatoren + und * können auf Zeichenketten angewandt werden

Beispiel: print(3 * 'mi' + 'mo')

Rechnen mit Zahlen

Addition +, Subtraktion -, Multiplikation *, Division /, Rest %, Division ohne Rest //

Potenzieren **

import math as m – importiert das Modul math und bindet daran den Namen m

Nun können wir die Wurzel ziehen print(m.sqrt(25))

4 Listen und Wörterbücher

Listen in Python – viele Inhalte geordnet speichern (Array in anderen Programmiersprachen)

Eckige Klammern verwenden

```
vornamen = ["Axel", "Elke", "Martin"]
```

`print(vornamen)` gibt alle Namen aus.

`print(vornamen[0])` gibt das erste Element der Liste aus.

`print(vornamen[0:2])` gibt die ersten beiden Elemente der Liste aus.

Merke: Der Index beginnt mit Null!

`print(vornamen[-1])` gibt das letzte Element der Liste aus.

`vornamen[2] = "Fritz"` überschreibt das dritte (und letzte) Element der Liste.

Listen durch weitere Elemente erweitern

```
vornamen += ["Heike", "Sabine"]
```

Das ist die Kurzschreibweise für `vornamen = vornamen + ["Heike", "Sabine"]`

Das erste Element aus der Liste entfernen

```
del vornamen[0]
```

Komplette Liste löschen

```
del(vornamen)
```

Methoden und Listen

Elemente an eine leere Liste anhängen

```
buchstaben = []
```

```
buchstaben.append("a")
```

```
buchstaben.append("b")
```

`print(buchstaben)` gibt aus ['a', 'b']

Elemente in Liste an bestimmte Position einfügen

```
buchstaben.insert(1, "c")
```

`print(buchstaben)` gibt aus ['a', 'c', 'b']

Element aus Liste entfernen anhand seines Wertes

```
buchstaben.remove("b")
```

`print(buchstaben)` gibt aus ['a', 'c']

Dictionary – Wörterbuch in Python (Zuordnungstabelle)

```
englisch_deutsch = {}
```

```
englisch_deutsch["cat"] = "Katze"
```

```
englisch_deutsch["dog"] = "Hund"
```

```
englisch_deutsch["cow"] = "Kuh"
```

```
englisch_deutsch["bird"] = "Vogel"
```

Inhalt ausgeben

`englisch_deutsch` gibt aus {'cat': 'Katze', 'dog': 'Hund', 'cow': 'Kuh', 'bird': 'Vogel'}

Bestimmtes Element des Dictionary erhalten

`englisch_deutsch["dog"]` gibt aus 'Hund'

Schlüssel des Dictionary

`englisch_deutsch.keys()` gibt aus `dict_keys(['cat', 'dog', 'cow', 'bird'])`

Werte des Dictionary

`englisch_deutsch.values()` gibt aus `dict_values(['Katze', 'Hund', 'Kuh', 'Vogel'])`

Wenn jeder Wert nur einmal im Dictionary vorkommt, können Schlüssel und Werte mit folgender Programmzeile vertauscht werden (Erläuterung später)

`deutsch_englisch = dict((v,k) for k,v in englisch_deutsch.items())`

`deutsch_englisch` gibt aus `{'Katze': 'cat', 'Hund': 'dog', 'Kuh': 'cow', 'Vogel': 'bird'}`

Element löschen

`del englisch_deutsch["cow"]`

`englisch_deutsch` gibt aus `{'cat': 'Katze', 'dog': 'Hund', 'bird': 'Vogel'}`

Tupel sind Listen, die nicht änderbar sind. Tupel schreibt man mit runden Klammern.

Zum Beispiel ein Wert im Koordinatensystem:

`punkt1 = (10, 22)`

Die Elemente von `punkt1` können nicht gelöscht oder überschrieben werden.

Tupel haben 2 Methoden `.count("gesucht")` und `.index("gesucht")`. Beispiel:

`zahlen = (10, 20, 23, 20, 54)`

`zahlen.count(20)` gibt 2 aus, weil die 20 zweimal vorkommt.

`zahlen.index(10)` gibt 0 aus, weil die 10 das erste Element ist.

5 Benutzereingaben anfordern

Input – Benutzereingaben anfordern

`benutzereingabe = input("Bitte Zahl eingeben ")`

Das Programm wartet, bis der Benutzer die Eingabe gemacht hat, z. B. 23

`print(benutzereingabe)` gibt '23' aus. Input liefert also Strings!

Zum Weiterrechnen müssen wir den String in eine Zahl umwandeln:

`benutzereingabe = int(benutzereingabe)`

`print(benutzereingabe)` gibt 23 aus. Damit können wir weiterrechnen.

Benutzereingabe verarbeiten

Einzelne Zeichen des Strings können nicht überschrieben werden. Deshalb mit Methoden arbeiten! Beispiel:

`vorname = input("Gib deinen Namen ein ")`

Das Programm wartet, bis der Benutzer die Eingabe gemacht hat, z. B. peter

Das Programm begrüßt den Benutzer mit seinem Vornamen. Vorne steht ein Großbuchstabe:

`print("Hallo " + vorname.capitalize())` gibt aus: Hallo Peter

Text ausgeben und 1 Sekunde warten, bevor es weitergeht:

`import time`

`print("Denke dir eine Zahl")`

`time.sleep(1)`

`print("Der Computer rät 50")`

6 Fallunterscheidungen

if-Abfrage in Python

Mit der if-Abfrage prüft das Programm, ob bestimmte Bedingungen erfüllt sind. Das Ergebnis (wahr oder falsch) beeinflusst den Programmablauf. Wichtig ist, dass nach der if-Abfrage alles das, was zur if-Abfrage gehört, eingerückt wird. Beispiel:

```
wert = 3
if wert < 5:
    print("Der Wert ist kleiner als 5")
    print("Ich gehöre auch noch zu der Bedingung")
print("und hier geht es nach der if-Abfrage weiter")
```

Eine Alternative angeben, wenn die Bedingung nicht erfüllt ist

```
if wert < 5:
    print("Der Wert ist kleiner als 5")
else:
    print("Der Wert ist gleich 5 oder größer als 5")
```

Alle Vergleichsoperatoren:

==	gleich
!=	ungleich
<	kleiner
>	größer
<=	kleiner oder gleich
>=	größer oder gleich

Mehrere Bedingungen prüfen – elif

```
wert = 9
if wert < 5:
    print('Wert ist kleiner als 5')
elif wert == 5:
    print('Wert ist exakt 5')
else:
    print('Wert ist größer als 5')
```

Es können beliebig viele elif Abfragen gemacht werden.

7 Schleifen

while-Schleife in Python

Über Schleifen können wir Programmschritte mehrmals ausführen lassen, bis eine festgelegte Bedingung erfüllt ist. Wichtig ist, dass alles das, was zur while-Schleife gehört, eingerückt wird. Im folgenden Beispiel zählen wir Durchgänge:

```
durchgang = 1
while durchgang < 11:
    print(durchgang)
    durchgang = durchgang + 1
print("nach der Schleife")
```

Ausgabe:

```
1
2
3
4
5
6
7
8
9
10
nach der Schleife
```

Die Anweisung break beendet die unendliche while-Schleife. Im folgenden Beispiel zählen wir Durchgänge:

```
durchgang = 1
while True:
    print(durchgang)
    durchgang += 1
    if durchgang > 10:
        break
print("nach der Schleife")
```

Ausgabe: wie oben

Die for-Schleife in Python arbeitet eine Liste ab. Wir können die Elemente der Liste direkt verwenden. Wichtig ist, dass alles das, was zur for-Schleife gehört, eingerückt wird.

Im folgenden Beispiel geben wir die Namen einer Liste aus.

```
vornamen = ["Axel", "Elke", "Martin"]
for element in vornamen:
    print(element)
print("nach der for-Schleife")
```

Ausgabe:

```
Axel
Elke
Martin
nach der for-Schleife
```

Die Funktion `range()` erstellt eine Liste:

```
folge = list(range(3))
print(folge)
```

Ausgabe: [0, 1, 2]

Wir können `range()` in der `for`-Schleife benutzen:

```
for durchgang in range(3):
    print(durchgang)
```

Ausgabe:

```
0
1
2
```

von-bis bei `range()`:

```
folge = list(range(2, 8))
print(folge)
```

Ausgabe: [2, 3, 4, 5, 6, 7]

Schrittweite bei `range()`:

```
folge = list(range(2, 8, 2))
print(folge)
```

Ausgabe: [2, 4, 6]

Eine Liste mit Dezimalzahlen erstellen wir mit einer `for`-Schleife:

```
dezimal = []
for i in range(10):
    dezimal.append(i * 0.3)
```

8 Einfacher Chatbot

Ein Chatbot ist ein Computer-Programm, mit dem man sich unterhalten kann. Wenn das Programm ein bekanntes Schlüsselwort erhält, gibt es eine passende Antwort. Fehlt ein bekanntes Schlüsselwort, antwortet das Programm mit einer allgemeinen Bemerkung.

Die Zufallsantworten stehen in einer Liste.

Die Reaktionsantworten stehen in einem Dictionary mit Schlüsselwort und Antwort.

Das Programm besteht aus einer Hauptschleife (`while benutzereingabe != "bye"`) und zwei eingeschlossenen Nebenschleifen.

Die erste Nebenschleife wiederholt die Frage solange, bis der Benutzer etwas eingibt.

Dann wird die Benutzereingabe in einzelne Wörter zerlegt.

Die zweite Nebenschleife durchsucht die Benutzereingabe nach bekannten Schlüsselwörtern.

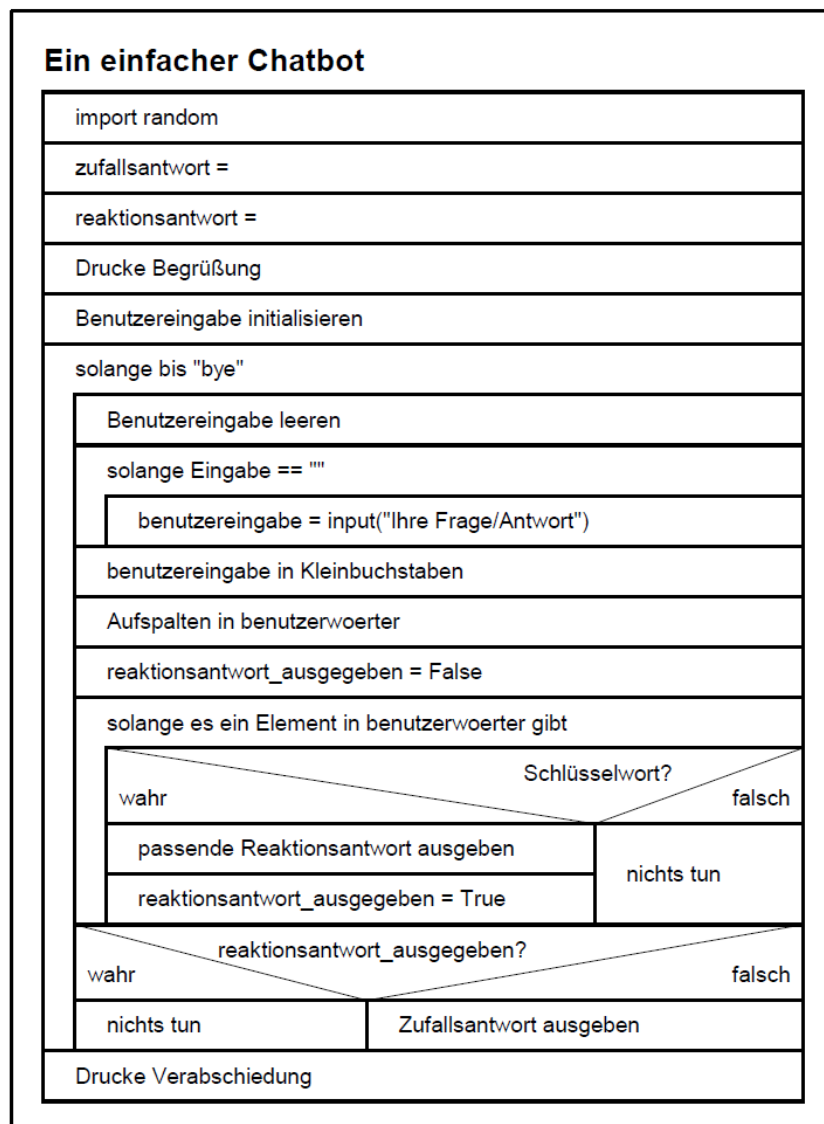
Wir können das Dictionary erweitern, damit wir uns mit dem Programm über Themen wie Kino, Busfahren usw. unterhalten können. Das Programm reagiert auf Wörter und nicht auf ganze Sätze. Es kann nicht auf Schlüssel der Form "Mir geht es gut" reagieren.

Der Programmablauf kann durch ein Flussdiagramm oder ein Nassi-Shneiderman-Diagramm (auch Struktogramm) beschrieben werden.

Flussdiagramm (© <https://www.python-lernen.de/>)



Struktogramm



9 Funktionen

Eine Funktion ist Programmcode, der gezielt aufgerufen wird. Wir schreiben eine Funktion für eine bestimmte Aufgabe. Der Programmcode wird dadurch übersichtlicher. Wenn wir eine Funktion mehrmals benötigen, rufen wir sie sich mehrmals auf. Wir definieren eine Funktion mit dem Schlüsselwort `def`:

```
def ausgabe():
    print("Ausgabe von Text aus einer Funktion")
```

Aufruf: `ausgabe()`

Ausgabe: Ausgabe von Text aus einer Funktion

Achtet auf die Klammern nach dem Funktionsnamen.

Wenn wir Werte in die Funktion übergeben, geben wir auch den Datentyp des Wertes an. Python prüft den Datentyp nicht. Aber es gibt Zusatzprogramme, die Datentypen prüfen können.

Durch die Angabe des Datentyps wird unser Programmcode besser lesbar! Beispiel:

```
def ausgabe(wert1: int):
    print("Wert1 = ", wert1)
```

Aufruf: `ausgabe(5)`

Ausgabe: Wert1 = 5

Wir können auch mehrere Werte übergeben. Beispiel:

```
def ausgabe(wert1: int, wert2: int):
    print("Wert1 = ", wert1, "Wert2 = ", wert2)
```

Aufruf: `ausgabe(5, 6)`

Ausgabe: Wert1 = 5 Wert2 = 6

Werte mit Vorgaben stehen rechts von den Werten ohne Vorgaben. Beispiel:

```
def ausgabe(wert1: int, wert2: int = 15):
    print("Wert1 = ", wert1, "Wert2 = ", wert2)
```

Aufruf: `ausgabe(5)`

Ausgabe: Wert1 = 5 Wert2 = 15

Aufruf: `ausgabe(5, 80)`

Ausgabe: Wert1 = 5 Wert2 = 80

Funktionen können einen Wert an das rufende Programm zurückgeben. Wir geben den Datentyp des Rückgabewerts an. Beispiel:

```
def verdoppeln(eingabewert: int) -> int:
    rueckgabewert = eingabewert * 2
    return rueckgabewert
```

Aufruf: `ergebnis = verdoppeln(5)`

`print(ergebnis)`

Ausgabe: 10

Die Variable `rueckgabewert` steht außerhalb der Funktion nicht zur Verfügung. Mit `return` geben wir den Wert der Variable `rueckgabewert` an das rufende Programm zurück.

Konstruktionsanleitung für Programme und Funktionen:

1. Kurzbeschreibung
2. Datenanalyse
3. Funktion definieren: Name – Eingabewert: Datentyp – Rückgabewert: Datentyp
4. Gerüst
5. Rumpf
6. Ergebnisse prüfen
7. Unittest

Aufgabe

	Grundgebühr pro Monat	Verbrauchspreis pro kWh
Stromtarif "Watt für wenig"	13,50 €	0,75 €

Schreibe ein Programm, das den Monatsverbrauch in Kilowattstunden akzeptiert und den monatlichen Rechnungsbetrag beim Tarif "Watt für wenig" berechnet.

Lösung

```

1 # monatliche Kosten für Tarif Watt für wenig berechnen (Schritt 1)
2
3 # Funktion mit Datentyp (Schritt 3 und Schritt 5)
4 def watt_fuer_wenig(verbrauch: float) -> float:
5     kosten = 13.5 + verbrauch * 0.75
6     return kosten
7
8 # Ergebnisse berechnen, Liste initialisieren (Ab hier: Schritt 6)
9 verbrauch = range(0, 150, 10)
10 kostenl = []
11
12 # Listen mit den Kosten des Tarifs erstellen
13 for x in verbrauch:
14     kostenl.append(watt_fuer_wenig(x))
15
16 # Ergebnisse drucken
17 print(" Verbrauch  Watt für wenig")
18 for i in range(len(verbrauch)):
19     print("{:10.2f} {:15.2f}".format(verbrauch[i], kostenl[i]))
20
21 # Modul für das Plotten von Graphen importieren
22 import matplotlib.pyplot as plt
23
24 # Ergebnisse plotten
25 plt.plot(verbrauch, kostenl)
26 plt.show()
27
    
```

10 Formatierte Ausgabe und Graphen plotten

Funktionen können eine Liste als Eingabe bekommen. Beispiel:

```
# Kosten für Volumentarif berechnen
# Die Kosten werden durch das Datenvolumen im Monat bestimmt

# Funktion mit Datentyp
def volumentarif(verbrauch: list[float]) -> float:
    kosten = 0
    for i in range(len(verbrauch)):
        kosten += verbrauch[i] * 6.5
    return kosten
```

Die Funktion bekommt den Verbrauch als list[float] und berechnet die Kosten.

Aufruf: volumentarif([1.2, 1.5, 2])

Ausgabe: 30.55

Eine Funktion kann eine Liste an das rufende Programm zurückgeben. Beispiel:

```
# Potenzen berechnen: 2. Potenz und 3. Potenz des Eingabewerts

# 2. und 3. Potenz berechnen
# Funktion mit Datentyp
def potenzen(zahl: float) -> list[float]:
    quadratzahl = zahl * zahl
    kubikzahl = quadratzahl * zahl
    return [quadratzahl, kubikzahl]

# Ergebnisse drucken
print("Zahl Quadratzahl Kubikzahl")
for zahl in range(4):
    ergebnis = potenzen(zahl)
    print("{:4.2f} {:12.2f} {:10.2f}".format(zahl, ergebnis[0], ergebnis[1]))
```

Die Funktion gibt die Potenzen als list[float] an das rufende Programm zurück.

Beim Ausdruck adressieren wir die Elemente der Liste mit ergebnis[0] und ergebnis[1].

Die Funktion "potenzen" ist nicht empfehlenswert. Es ist besser, für jede Aufgabe eine Funktion zu schreiben. Das folgende Beispiel ist besser:

```
# Potenzen berechnen: 2. Potenz und 3. Potenz des Eingabewerts

# 2. Potenz berechnen
# Funktion mit Datentyp
def quadratzahl(zahl: float) -> float:
    quadratzahl = zahl * zahl
    return quadratzahl

# 3. Potenz berechnen
# Funktion mit Datentyp
def kubikzahl(zahl: float) -> float:
    kubikzahl = zahl * zahl * zahl
    return kubikzahl

# Ergebnisse drucken
print("Zahl Quadratzahl Kubikzahl")
for zahl in range(4):
    print("{:4.2f} {:12.2f} {:10.2f}".format(zahl, quadratzahl(zahl), kubikzahl(zahl)))
```

Wir erstellen eine Funktion zur Umrechnung von Britischen Pfund in Euro nach der Konstruktionsanleitung auf Seite 10. Die Vorlage `meine_Aufgabe.py` hilft uns dabei:

```

1 # Aufgabe der Funktion ...
2
3 # Input der Funktion ...
4 # Output der Funktion ...
5
6 # Funktion mit Datentyp
7 def meine_funktion(mein_input: datentyp) -> datentyp:
8     mein_output = ...
9     return mein_output
10
11 # Input und Output initialisieren
12 meine_input_liste = ...
13 meine_output_liste = []
14
15 # Output erstellen
16 for x in meine_input_liste:
17     meine_output_liste.append(meine_funktion(x))
18
19 # Ergebnisse drucken
20 print("Meine Überschrift")
21 for i in range(len(meine_input_liste)):
22     print("Mein Format String".format(meine_input_liste[i], meine_output_liste[i]))
    
```

Wir kopieren die Vorlage und geben ihr einen passenden Namen, z. B. `gbp_in_eur.py`. Dann machen wir gezielte Änderungen in der Datei `gbp_in_eur.py`:

- An den Stellen mit ... ergänzen wir die Beschreibung bzw. den Code für unsere Aufgabe.
- Die Namen `meine_funktion`, `mein_input` usw. ersetzen wir durch treffende Namen.

Wir prüfen die Ergebnisse mit Hilfe von Input-Liste und Output-Liste. Den Inhalt der Input-Liste entnehmen wir der Aufgabenstellung. Den Inhalt der Output-Liste erhalten wir durch den Aufruf unserer Funktion in einer Schleife. In einer Tabelle stellen wir Input-Liste und Output-Liste nebeneinander.

```

1 # Aufgabe der Funktion ist die Umrechnung von britischen Pfund in Euro
2
3 # Input der Funktion ist eine Dezimalzahl in der Einheit britische Pfund
4 # Output der Funktion ist eine Dezimalzahl in der Einheit Euro
5 # Umrechnungsfaktor 1 gbp = 1.21 eur
6
7 # Funktion mit Datentyp
8 def gbp_in_eur(gbp: float) -> float:
9     eur = 1.21 * gbp
10    return eur
11
12 # Input und Output initialisieren
13 gbp_liste = []
14 for i in range(21):
15     gbp_liste.append(i * 0.5)
16 eur_liste = []
17
18 # Output erstellen
19 for x in gbp_liste:
20     eur_liste.append(gbp_in_eur(x))
21
22 # Ergebnisse drucken
23 print("gbp    eur")
24 for i in range(len(gbp_liste)):
25     print("{:5.2f} {:5.2f}".format(gbp_liste[i], eur_liste[i]))
    
```

Eine schöne Tabelle erhalten wir durch den Formatstring der Funktion `print()` in Zeile 25. Die Platzhalter `{:5.2f}` sorgen dafür, dass die Eingabewerte in einem Feld der Breite 5 mit 2 Nachkommastellen als Fließkommazahl ausgegeben werden:

GBP	EUR
0.00	0.00
0.50	0.60
1.00	1.21

11 unittest

Bisher haben wir ein Programm oder eine Funktion geprüft, indem wir die formatierte Ausgabe oder den Graphen angeschaut und mit Sollwerten verglichen haben. Diese Prüfung müssten wir bei jeder Änderung des Programms wiederholen.

Es gibt eine bessere Möglichkeit: Wir können die Prüfung mit Hilfe des Moduls „unittest“ automatisieren. Dabei prüfen wir, ob das Funktionsergebnis und der Sollwert gleich sind (equal).

Die Vorlage Mein_Unittest.py hilft uns dabei:

```
1 # Modul für den unittest importieren
2 import unittest
3
4 # zu testende Funktion einfügen
5 def meine_funktion:
6     ...
7
8 # Testfunktionen definieren
9 class Mein_test(unittest.TestCase):
10
11     def test_meine_funktion(self):
12         self.assertEqual(meine_funktion(mein_input1), mein_output1)
13         self.assertEqual(meine_funktion(mein_input2), mein_output2)
14         self.assertEqual(meine_funktion(mein_input3), mein_output3)
15         self.assertEqual(meine_funktion(mein_input4), mein_output4)
16
17 # unittest ausführen, wenn die Datei direkt aufgerufen wird
18 if __name__ == '__main__':
19     unittest.main()
```

Wir kopieren die Vorlage und geben ihr einen passenden Namen, z. B. Hunderter_Unittest.py. Dann machen wir gezielte Änderungen in der Datei Hunderter_Unittest.py:

Zeile 5: `def meine_funktion` ersetzen wir durch eine Kopie der Funktion, die wir testen wollen.

Zeile 9: `Mein_test` wir durch einen treffenden Namen.

Zeile 11: `test_meine_funktion` ersetzen wir durch `test_` und den Funktions-Namen.

Zeile 12-15: `meine_funktion` ersetzen wir durch den Funktions-Namen.

`mein_input` und `mein_output` ersetzen wir durch Paare von Input und Sollwert.

```
1 # Modul für den unittest importieren
2 import unittest
3
4 # zu testende Funktion einfügen
5 def hunderter(ganzzahl: int) -> int:
6     anzahl = ganzzahl // 100
7     return anzahl
8
9 # Testfunktionen definieren
10 class Hunderter_test(unittest.TestCase):
11
12     def test_hunderter(self):
13         self.assertEqual(hunderter(0), 0)
14         self.assertEqual(hunderter(10), 0)
15         self.assertEqual(hunderter(100), 1)
16         self.assertEqual(hunderter(109), 1)
17         self.assertEqual(hunderter(200), 2)
18         self.assertEqual(hunderter(290), 2)
19         self.assertEqual(hunderter(900), 9)
20         self.assertEqual(hunderter(999), 9)
21
22 # unittest ausführen, wenn die Datei direkt aufgerufen wird
23 if __name__ == '__main__':
24     unittest.main()
```

Wenn wir Hunderter_Unittest.py starten, erhalten wir folgende Ausgabe:

```
.
-----
Ran 1 test in 0.069s

OK
```

Das bedeutet: 1 Test – nämlich die 8 Zeilen von test_hunderter() – hatte das Ergebnis **OK**.

Wenn die Paare von Input und Sollwert nicht zusammenpassen, erhalten wir eine Fehlermeldung:

```
F
=====
FAIL: test_hunderter (__main__.Hunderter_test)
-----
Traceback (most recent call last):
  File "C:\Users\marti\Documents\Python\11_Unittest\Meine_Hunderter_Unittest.py",
    line 17, in test_hunderter
      self.assertEqual(hunderter(200), 1)
AssertionError: 2 != 1

-----
Ran 1 test in 0.065s
```

FAILED (failures=1)

Die Fehlermeldung zeigt den ersten gefundenen Fehler und den Unterschied Istwert – Sollwert.

Die Vorlage Mein_Unittest.py arbeitet mit einer Kopie der Funktion, die wir testen wollen.

Besser ist, das Original der Funktion zu importieren und zu testen.

Die Vorlage Mein_Unittest2.py hilft uns dabei:

```
1 | # Modul für den Unittest importieren
2 | import unittest
3 |
4 | # Modul mit Zugriff auf die Variable sys.path importieren
5 | import sys
6 |
7 | # Pfad der zu testenden Funktionen anhängen
8 | sys.path.append("mein Pfad")
9 |
10 | # zu testende Funktion importieren
11 | import meine_funktion
12 |
13 | # Testfunktionen definieren
14 | class Mein_test(unittest.TestCase):
15 |
16 |     def test_meine_funktion(self):
17 |         self.assertEqual(meine_funktion.meine_funktion(mein_input1), mein_output
18 |         self.assertEqual(meine_funktion.meine_funktion(mein_input2), mein_output
19 |         self.assertEqual(meine_funktion.meine_funktion(mein_input3), mein_output
20 |         self.assertEqual(meine_funktion.meine_funktion(mein_input4), mein_output
21 |
22 | # Unittest ausführen, wenn die Datei direkt aufgerufen wird
23 | if __name__ == '__main__':
24 |     unittest.main()
```

Wir kopieren die Vorlage und geben ihr einen passenden Namen, z. B. billig_strom_unittest.py. Dann machen wir gezielte Änderungen in der Datei billig_strom_unittest.py:

Zeile 8: **mein Pfad** ersetzen wir durch den Datei-Pfad der Funktion, die wir testen wollen.

Zeile 11: **meine_funktion** ersetzen wir durch den Datei-Namen der Funktion, die wir testen wollen.

Zeile 9: **Mein_test** wir durch einen treffenden Namen.

Zeile 11: **test_meine_funktion** ersetzen wir durch **test_** und den Funktions-Namen.

Zeile 12-15: **meine_funktion.meine_funktion** ersetzen wir durch Datei-Namen.Funktions-Namen.
mein_input und **mein_output** ersetzen wir durch Paare von Input und Sollwert.

Wenn wir `billig_strom_unittest.py` laufen lassen, erhalten wir folgende Fehlermeldung:

```
F
=====
FAIL: test_billig_strom (__main__.Stromtarif_test)
=====
Traceback (most recent call last):
  File "C:\Users\marti\Documents\Python\11_Unittest\billig_strom_unittest.py", line
  19, in test_billig_strom
    self.assertEqual(billig_strom.billig_strom(20), 25.4)
AssertionError: 25.400000000000002 != 25.4

-----
Ran 1 test in 0.010s

FAILED (failures=1)
```

In der Fehlermeldung steht, dass Istwert und Sollwert von Fließkommazahlen nicht gleich sind.

Das liegt daran, dass die Fließkommazahlen im Computer immer Näherungen sind.

Wenn die Funktion, die wir testen wollen, mit Fließkommazahlen arbeitet, können wir nur prüfen, ob der Istwert und der Sollwert ungefähr gleich sind (almost equal).

Die Methode `assertAlmostEqual()` berechnet die Differenz zwischen Istwert und Sollwert. Die Differenz wird auf die 7. Stelle nach dem Komma gerundet und mit Null verglichen.

Wenn wir das Programm `billig_strom_unittest.py` mit der Methode `assertAlmostEqual()` arbeiten lassen, ist das Ergebnis OK.

```
1 # Modul für den Unittest importieren
2 import unittest
3
4 # Modul mit Zugriff auf die Variable sys.path importieren
5 import sys
6
7 # Pfad der zu testenden Funktionen anhängen
8 sys.path.append("../9_Funktionen")
9
10 # zu testende Funktion importieren
11 import billig_strom
12
13 # Testfunktionen definieren
14 class Stromtarif_test(unittest.TestCase):
15
16     def test_billig_strom(self):
17         self.assertAlmostEqual(billig_strom.billig_strom(0), 9.2)
18         self.assertAlmostEqual(billig_strom.billig_strom(10), 17.3)
19         self.assertAlmostEqual(billig_strom.billig_strom(20), 25.4)
20         self.assertAlmostEqual(billig_strom.billig_strom(50), 49.7)
21         self.assertAlmostEqual(billig_strom.billig_strom(100), 90.2)
22         self.assertAlmostEqual(billig_strom.billig_strom(140), 122.6)
23
24 # Unittest ausführen, wenn die Datei direkt aufgerufen wird
25 if __name__ == '__main__':
26     unittest.main()
27
```

12 Zweidimensionale Arrays

Mit Hilfe eines zweidimensionalen Arrays erstellen wir eine Matrix mit Spalten und Zeilen.

Beispiel: Eine Matrix mit 5 Zeilen und 8 Spalten

	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5								

In Python werden zweidimensionale Arrays als Listen von Listen erstellt:

```
# Matrix erzeugen: Liste von Listen

# Modul für den schönen Ausdruck (pretty print) importieren
import pprint

# Erzeugt eine Liste, die 5 Listen mit je 8 Elementen enthält
# Füllt alle Elemente mit "_"
breite = 8 # Buchstaben A bis H
hoehe = 5 # Zahlen 1 bis 5
Matrix = [["_"] for x in range(breite)] for y in range(hoehe)]

# Drucke die Matrix
print("\nInitialisierung")
pprint.pprint(Matrix)
```

Ausgabe:

```
Initialisierung
[['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_']]
```

Wir adressieren die Elemente der Matrix über zwei Indizes.

Der erste Index adressiert die Zeilen.

```
# Zeilen adressieren
print("\n1. Element jeder Zeile beschreiben")
Matrix[0][0] = "A1"
Matrix[1][0] = "A2"
Matrix[2][0] = "A3"
Matrix[3][0] = "A4"
Matrix[4][0] = "A5"
```

Ausgabe:

```
1. Element jeder Zeile beschreiben
[['A1', '_', '_', '_', '_', '_', '_', '_'],
 ['A2', '_', '_', '_', '_', '_', '_', '_'],
 ['A3', '_', '_', '_', '_', '_', '_', '_'],
 ['A4', '_', '_', '_', '_', '_', '_', '_'],
 ['A5', '_', '_', '_', '_', '_', '_', '_']]
```

Der zweite Index adressiert die Spalten.

```
# Spalten adressieren
print("\n1. Element jeder Spalte beschreiben")
Matrix[0][0] = "A1"
Matrix[0][1] = "B1"
Matrix[0][2] = "C1"
Matrix[0][3] = "D1"
Matrix[0][4] = "E1"
Matrix[0][5] = "F1"
Matrix[0][6] = "G1"
Matrix[0][7] = "H1"
```

Ausgabe:

```
1. Element jeder Spalte beschreiben
[['A1', 'B1', 'C1', 'D1', 'E1', 'F1', 'G1', 'H1'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_']]
```


Wenn wir einen langen String in ein Element dieses zweidimensionalen Arrays schreiben ...

```
# Langen String ins letzte Element schreiben - bleibt erhalten!
print("\nLangen String schreiben")
Matrix[4][7] = "langer String"
```

... dann vergrößert Python das Element, damit der lange String Platz findet. Ausgabe:

Langen String schreiben

```
[['A1', 'B1', 'C1', 'D1', 'E1', 'F1', 'G1', 'H1'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_', '_', 'langer String']]
```

Mehrdimensionale Arrays, die wie oben erstellt werden, sind in der Verarbeitung langsam.

Das Modul numpy erzeugt mehrdimensionale Arrays, die schnell verarbeitet werden.

Außerdem hat numpy viele Methoden, die den Umgang mit mehrdimensionalen Arrays vereinfachen. So wird ein zweidimensionales Array mit numpy erstellt:

```
# Matrix erzeugen mit numpy: nd array Objekt
# numpy arbeitet schnell und bietet viele nützliche Funktionen
```

```
# Modul für die Bearbeitung von Arrays importieren
import numpy as np
```

```
# Erzeugt ein Array mit 5 Zeilen und 8 Spalten
# Füllt alle Elemente mit "_"
# Bestimmt den Datentyp aus dem Füllwert
Matrix = np.full((5, 8), "_")
```

```
# Drucke die Matrix
print("\nInitialisierung")
print(Matrix)
```

```
# Drucke den Datentyp
print("Datentyp", Matrix.dtype)
```

Ausgabe:

Initialisierung

```
[['_', '_', '_', '_', '_', '_', '_', '_']
 ['_', '_', '_', '_', '_', '_', '_', '_']
 ['_', '_', '_', '_', '_', '_', '_', '_']
 ['_', '_', '_', '_', '_', '_', '_', '_']
 ['_', '_', '_', '_', '_', '_', '_', '_']]
```

Datentyp <U2

Python bestimmt Datentyp aus unserem Füllwert "_". Das ist ein String mit 2 Unicode-Zeichen.

Wir adressieren die Elemente der Matrix über zwei Indices.

Der erste Index adressiert die Zeilen.

```
Matrix[0, 0] = "A1"
Matrix[1, 0] = "A2"
```

Der zweite Index adressiert die Spalten.

```
Matrix[0, 0] = "A1"
Matrix[0, 1] = "B1"
```

Bei einem numpy Array können wir die Indizes durch ein Komma trennen. Wir sparen zwei eckige Klammern.

Wenn wir einen langen String in ein Element des zweidimensionalen numpy Arrays schreiben ...

```
# Langen String ins letzte Element schreiben - wird abgeschnitten!
print("\nlangen String schreiben")
Matrix[4, 7] = "langer String"
```

... dann verkürzt Python den String auf die Länge des Elements. Ausgabe:

langen String schreiben

```
[['A1', 'B1', 'C1', 'D1', 'E1', 'F1', 'G1', 'H1']
 ['_', '_', '_', '_', '_', '_', '_', '_']
 ['_', '_', '_', '_', '_', '_', '_', '_']
 ['_', '_', '_', '_', '_', '_', '_', '_']
 ['_', '_', '_', '_', '_', '_', '_', 'la']]
```

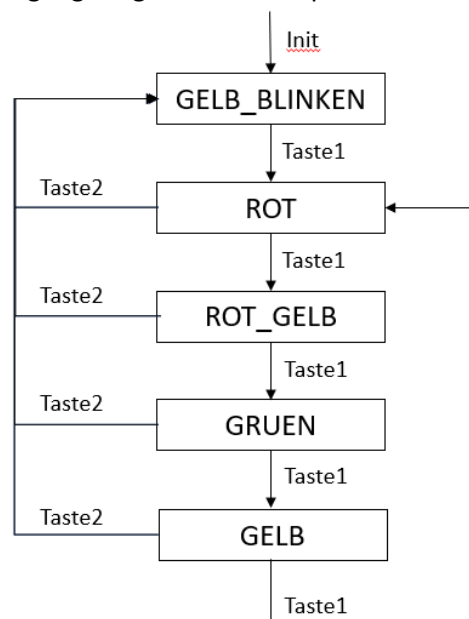
13 Zustandsmaschine

Auch Mikrocontroller können wir mit Python programmieren. Dabei läuft auf dem Mikrocontroller MicroPython. MicroPython ist eine schlanke und effiziente Implementierung der Python 3 Programmiersprache, die einen kleinen Teil der Python Standardbibliothek enthält und für den Betrieb auf einem Mikrocontroller optimiert ist. Mikrocontroller werden sehr oft mit Hilfe von Zustandsmaschinen programmiert.

Eine Zustandsmaschine ist ein Modell eines Verhaltens, bestehend aus Zuständen, Zustandsübergängen und Aktionen. Eine Zustandsmaschine wird durch ein Zustandsübergangsdiagramm beschrieben. Zustände werden durch Rechtecke symbolisiert. Zustandsübergänge werden durch Pfeile dargestellt, die jeweils zwei Zustände verbinden. Die Bedingungen für die Übergänge schreibt man neben die Pfeile. Alternativ kann eine Zustandsmaschine durch eine Übergangstabelle beschrieben werden.

Beispiel: Das Modell einer Ampel hat die Zustände: GELB_BLINKEN, ROT, ROT_GELB, GRUEN, GELB. Die Zustandsübergänge werden durch Steuersignale ausgelöst. In unserem Modell sind das die Tasten „Taste1“ und „Taste2“.

Wir zeichnen das Zustandsübergangsdiagramm der Ampel:



Alternativ beschreiben wir die Ampel durch ihre Übergangstabelle:

Zustandsübergang		Bedingung
Init	→ GELB_BLINKEN	-
GELB_BLINKEN	→ ROT	Taste1
ROT	→ ROT_GELB	Taste1
ROT_GELB	→ GRUEN	Taste1
GRUEN	→ GELB	Taste1
GELB	→ ROT	Taste1
ROT	→ GELB_BLINKEN	Taste2
ROT_GELB	→ GELB_BLINKEN	Taste2
GRUEN	→ GELB_BLINKEN	Taste2
GELB	→ GELB_BLINKEN	Taste2

Zu Beginn (Init) hat unser Modell den Zustand GELB_BLINKEN. Mit Taste1 durchläuft unser Modell die Zustände ROT – ROT_GELB – GRUEN – GELB – ROT usw. Durch Taste1 kehrt unser Modell nicht in den Zustand GELB_BLINKEN zurück. Dafür ist Taste2 zuständig. Durch Taste2 wechselt unser Modell aus jedem Zustand in den Zustand GELB_BLINKEN.

Unser Mikrocontroller soll das Verhalten einer Ampel nachbilden. Zwei Bibliotheken helfen uns dabei:

- Mit den Funktionen der Bibliothek MicroPython Statemachine beschreiben wir die Zustände und die Zustandsübergänge.
- Mit den Funktionen der Bibliothek MicroPython Neotimer blinken wir mit der Ampel-LED und entprellen wir die Tasten. (Die Betätigung einer elektromechanischen Taste führt kurzzeitig zu einem mehrfachen Öffnen und Schließen des Kontakts. Das nennt man „prellen“. Eine entprellte Taste öffnet und schließt den Kontakt nur einmal.)

Im Mikrocontroller-Programm ist Folgendes enthalten:

1. Kurzbeschreibung
2. Import der notwendigen Bibliotheken
3. Definition der Pins der Ampel LEDs und der Pins der Tasten
4. Objekt state_machine wird erzeugt
5. Timer werden erzeugt
6. Definition der Funktionen in den Zuständen
7. Definition der Funktionen zur Steuerung der Zustandsübergänge
8. Definition der Zustände
9. Definition der Zustandsübergänge
10. Loop

Die Punkte 6, 7, 8 und 9 setzen das Zustandsübergangsdiagramm (oder die Übergangstabelle) in Code um.

Zu 6: Funktion im Zustand GELB_BLINKEN:

```
43 # Die Ampel blinkt gelb
44 def gelb_blinken():
45     led_rot.value(0)
46     led_gruen.value(0)
47     if myTimer_500.repeat_execution():
48         led_gelb.value(not led_gelb.value())
```

Zu 7: Funktion zur Steuerung der Zustandsübergänge durch Taste1

```
74 # Taste1 gedrückt?
75 def taste1_gedrueckt():
76     if myTimer_250.debounce_signal(taste1.value() == 0):
77         return True
78     else:
79         return False
```

Zu 8: Definition der Zustände

```
88 # Definition der Zustände
89 GELB_BLINKEN = state_machine.add_state(gelb_blinken)
90 ROT = state_machine.add_state(rot)
91 ROT_GELB = state_machine.add_state(rot_gelb)
92 GRUEN = state_machine.add_state(gruen)
93 GELB = state_machine.add_state(gelb)
```

Zu 9: Definition der Zustandsübergänge durch Taste1

```
95 # Zustandsübergänge hinzufügen
96 # Übergänge durch Tasten
97 GELB_BLINKEN.attach_transition(taste1_gedrueckt, ROT)
98 ROT.attach_transition(taste1_gedrueckt, ROT_GELB)
99 ROT_GELB.attach_transition(taste1_gedrueckt, GRUEN)
00 GRUEN.attach_transition(taste1_gedrueckt, GELB)
01 GELB.attach_transition(taste1_gedrueckt, ROT)
```

Zu 10: Loop

```
116 # Loop
117 while True:
118     state_machine.run()
```

Die Funktion `run()` des `state_machine` Objekts wird also in einer unendlichen Schleife aufgerufen. Dadurch führt das Programm solange den Code des aktuellen Zustands aus, bis eine Bedingung für einen Zustandsübergang wahr wird. Mikrocontroller-Programme werden sehr oft in einer unendlichen Schleife ausgeführt.

Auch auf unserem PC können wir eine Zustandsmaschine programmieren. Hier die Unterschiede zur Programmierung des Mikrocontrollers:

- Die Bibliothek MicroPython StateMachine (`statemachine.py`) können wir ohne Änderung einsetzen.
- Bei der Bibliothek MicroPython Neotimer (`neotimer_win.py`) musste nur der Zugriff auf die Rechner-Uhr geändert werden.
- Anstelle von LEDs und Tasten verwenden wir die Tasten unserer Tastatur und machen `print`-Ausgaben auf den Bildschirm.

Die Funktion im Zustand GELB_BLINKEN sieht so aus:

```
34 # Funktionen in den Zuständen
35 # Bis zum Zustandswechsel wird die Funktion wiederholt aufgerufen
36 # Einmaliger Aufruf durch Abfrage von state_machine.execute_once
37 # Ampel blinkt gelb
38 def gelb_blinken():
39     if state_machine.execute_once:
40         print("Die Ampel blinkt gelb")
41     if mytimer_500.repeat_execution():
42         print("gelb blinken() aufgerufen")
```

Wir schreiben eine Funktion für die Steuerung der Zustandsübergänge durch eine beliebige Taste.

Dazu verwenden wir die Funktion `kbhit()` aus der Bibliothek für Windows-Funktionen:

```
64 # Tasten einlesen
65 def taste_gedruickt(zeichen):
66     global key_in
67     # Taste gedrückt?
68     if msvcrt.kbhit():
69         # Ja: Zeichen einlesen
70         key_in = msvcrt.getwch()
71     # Taste für den Zustandsübergang?
72     if key_in == zeichen:
73         # globale Variable zurücksetzen
74         key_in = ' '
75         return True
76     else:
77         return False
```

Mit ihrer Hilfe schreiben wir die Funktion für die Steuerung der Zustandsübergänge durch Taste '1':

```
79 def tastel_gedruickt():
80     return taste_gedruickt('1')
```

Definition der Zustände:

```
85 # Zustände definieren
86 # Initial ist der erste Zustand
87 GELB_BLINKEN = state_machine.add_state(gelb_blinken)
88 ROT = state_machine.add_state(rot)
89 ROT_GELB = state_machine.add_state(rot_gelb)
90 GRUEN = state_machine.add_state(gruen)
91 GELB = state_machine.add_state(gelb)
```

Definition der Zustandsübergänge durch Taste '1':

```
93 # Zustandsübergänge hinzufügen
94 # Normale Übergänge
95 GELB_BLINKEN.attach_transition(tastel_gedruickt, ROT)
96 ROT.attach_transition(tastel_gedruickt, ROT_GELB)
97 ROT_GELB.attach_transition(tastel_gedruickt, GRUEN)
98 GRUEN.attach_transition(tastel_gedruickt, GELB)
99 GELB.attach_transition(tastel_gedruickt, ROT)
```

Im Loop wird die Funktion run() des state_machine Objekts in einer unendlichen Schleife aufgerufen. Außerdem verarbeiten wir dort die Taste 'q' für den Abbruch:

```
114 # Loop
115 while True:
116     state_machine.run()
117     # key_in wird in der Funktion taste_gedrueckt() gesetzt
118     # Abbruch mit der Taste q
119     if key_in == 'q':
120         break
```

WICHTIG:

Wenn wir das Programm in IDLE starten, gibt es einen Konflikt zwischen .mainloop() und .kbhit() und das Programm arbeitet nicht richtig.

Deshalb muss das Programm von der Eingabeaufforderung aus gestartet werden.

Hier die Ausgabe des Programms. Es wurde 5x die Taste '1' gedrückt. Danach die Taste '2' und dann die Taste 'q'.

Zustandsautomat zur Steuerung einer Ampel. Zustandswechsel mit den Tasten 1 und 2

Abbruch mit der Taste q

```
Die Ampel blinkt gelb
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
Die Ampel ist rot
Die Ampel ist rot/gelb
Die Ampel ist grün
Die Ampel ist gelb
Die Ampel ist rot
Die Ampel blinkt gelb
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
gelb_blinken() aufgerufen
```

14 GUI mit tkinter

Ein GUI (Graphical User Interface) erstellen wir mit Hilfe des Moduls tkinter. Es gehört zur Standard-Installation von Python. Beispiel:

```

1 # GUI über das Modul tkinter
2 # Demonstration des Label-Widgets
3
4 # Modul für die GUI-Erstellung importieren
5 import tkinter as tk
6
7 # Konstruktor für das Fenster aufrufen
8 root = tk.Tk()
9 root.title("Label Widget Demo")
10
11 # Breite und Höhe des Fensters festlegen
12 root.geometry("320x200")
13
14 # Textausgabe erzeugen
15 label1 = tk.Label(root, text="Hallo Welt")
16
17 # in GUI Elemente einbetten
18 label1.pack(side="left")
19
20 # Bildausgabe erzeugen
21 bild1 = tk.PhotoImage(file="biene.png")
22 label2 = tk.Label(root, image=bild1)
23
24 # in GUI Elemente einbetten
25 label2.pack(side="right")
26
27 # Hauptschleife, damit die GUI angezeigt wird
28 root.mainloop()
    
```

Erklärung:

Wir importieren tkinter (Zeile 5), erstellen ein Fenster (Zeile 8) und setzen einen Fenstertitel (Zeile 9).

Wir setzen die Dimensionen des Fensters (Zeile 12). Passiert automatisch, wenn die Zeile fehlt.

Wir erstellen die Widgets (Window Gadgets) Textausgabe (Zeile 15) und Bildausgabe (Zeile 21, 22).

In Zeile 18 und Zeile 25 platzieren wir die Widgets mit Hilfe der Methode pack().

Der Aufruf in Zeile 28 ist notwendig, damit das Fenster angezeigt wird, wenn wir das Programm durch einen Doppelklick im Windows-Explorer starten.

Mit Hilfe der Methode grid() können wir die Gadgets gezielt in einem (gedachten) Gitter im Fenster platzieren. Beispiel:

```

# im Gitter platzieren
label1.grid(row=0, column=0)
    
```

Zeilenhöhe und Spaltenbreite werden automatisch eingestellt. Alternativ können wir mit den Methoden rowconfigure() und columnconfigure() minimale Größen im Fenster einstellen. Beispiel:

```

# Gitter konfigurieren
# 7 Zeilen, 80 Pixel hoch
for i in range(7):
    root.rowconfigure(i, minsize=80)
# 11 Spalten, 80 Pixel breit
for i in range(11):
    root.columnconfigure(i, minsize=80)
    
```

tk.Label() akzeptiert viele Optionen zum Text. Mit fg (foreground) und bg (background) legen wir Farben fest. Mit font legen wir die Schriftart und die Schriftgröße fest. Beispiel:

```

# Textausgabe erzeugen
label2 = tk.Label(root, text="R1 / C1", bg="lightgreen", font = ("Arial", "24"))
    
```

Eine Schaltfläche erzeugen wir mit `tk.Button()`. Beispiel:

```

1 | # GUI über das Modul tkinter
2 | # Demonstration von Schaltflächen
3 |
4 | # Modul für die GUI-Erstellung importieren
5 | import tkinter as tk
6 |
7 | # Funktion für Schaltfläche 1
8 | def aktionSF1():
9 |     # Steuervariable auslesen und neu setzen
10 |     zaehler.set(zaehler.get() + 1)
11 |
12 | # Funktion für Schaltfläche 2
13 | def aktionSF2():
14 |     # Steuervariable auslesen und neu setzen
15 |     zaehler.set(zaehler.get() - 1)
16 |
17 | # Konstruktor für das Fenster aufrufen
18 | root = tk.Tk()
19 |
20 | # Textausgabe erzeugen und im Gitter platzieren
21 | label1 = tk.Label(root, text="Hallo Welt", bg="orange", font = ("arial", 25))
22 | label1.grid(row = 0, column = 1)
23 |
24 | # Steuervariable erzeugen und initialisieren
25 | zaehler = tk.IntVar()
26 | zaehler.set(0)
27 |
28 | # Textausgabe erzeugen und im Gitter platzieren
29 | # Der Wert der Steuervariable wird angezeigt
30 | label2 = tk.Label(root, textvariable=zaehler, bg="green", font = ("arial", 25))
31 | label2.grid(row = 2, column = 1)
32 |
33 | # Schaltfläche erzeugen und im Gitter platzieren
34 | schaltf1 = tk.Button(root, text="größer", font = ("arial", 25), command=aktionSF1)
35 | schaltf1.grid(row = 1, column = 0)
36 |
37 | # Schaltfläche erzeugen und im Gitter platzieren
38 | schaltf2 = tk.Button(root, text="kleiner", font = ("arial", 25), command=aktionSF2)
39 | schaltf2.grid(row = 1, column = 2)
40 |
41 | # Hauptschleife, damit die GUI angezeigt wird
42 | root.mainloop()

```

Die beiden Widgets Schaltfläche (Zeile 34 und Zeile 38) benötigen jeweils eine Funktion, die ausgeführt wird, wenn die Schaltfläche betätigt wird. Im Beispiel sind es die Funktionen:

`command = actionSF1`

`command = actionSF2`

Diese Funktionen werden oben im Programm definiert (Zeilen 8, 9, 10, 13, 14, 15), damit sie bekannt ist, bevor die Schaltfläche erzeugt wird.

Wenn wir in einer Funktion eine Variable definieren, ist diese "lokal" d. h. nur innerhalb der Funktion bekannt. Hier hilft uns die Steuervariable von tkinter, die wir im Hauptprogramm erzeugen (Zeile 25).

Mehre Widgets teilen sich eine Steuervariable und die Steuervariable "kennt" diese Widgets.

Das bedeutet:

Wenn unser Programm mit der Methode `zaehler.set()` einen Wert in die Steuervariable schreibt, werden alle mit der Steuervariable verbundenen Widgets aktualisiert.

Mit der Steuervariable `zaehler` können unsere Funktion `aktionSF1` und `aktionSF2` die Textausgabe über `zaehler.get()` und `zaehler.set()` steuern.

Ein Eingabefeld erzeugen wir mit `tk.Entry()`. Beispiel:

```

1 # Umrechnung von Grad Celsius in Grad Kelvin
2
3 # Modul für die GUI-Erstellung importieren
4 import tkinter as tk
5
6 # Konstruktor für das Fenster aufrufen
7 root = tk.Tk()
8 root.title("Umrechnung Celsius in Kelvin")
9
10 # Funktion zur Umrechnung von Celsius in Kelvin
11 def celsius_in_kelvin():
12     # Eingabefeld auslesen und in Integer umwandeln
13     celsius = int(eingabefeld_wert.get())
14     # Umrechnen
15     kelvin = celsius + 273
16     # Steuervariable setzen
17     ausgabefeld_wert.set(kelvin)
18
19 # Steuervariable für das Eingabefeld erzeugen
20 eingabefeld_wert=tk.StringVar()
21
22 # Eingabefeld erzeugen und in GUI Elemente einbetten
23 eingabefeld = tk.Entry(root, textvariable=eingabefeld_wert, font = ("arial", 25))
24 eingabefeld.pack()
25
26 # Steuervariable für das Ausgabefeld erzeugen und initialisieren
27 ausgabefeld_wert = tk.StringVar()
28 ausgabefeld_wert.set(" ")
29
30 # Ausgabefeld erzeugen und in GUI Elemente einbetten
31 ausgabefeld = tk.Label(root, textvariable=ausgabefeld_wert, bg="yellow", font = ("arial"
32 ausgabefeld.pack()
33
34 # Schaltfläche erzeugen und im Gitter platzieren
35 schaltfl = tk.Button(root, text="Aktion durchführen", command=celsius_in_kelvin, bg="lig
36 schaltfl.pack()
37
38 # Hauptschleife, damit die GUI angezeigt wird
39 root.mainloop()
    
```

Das Widget Eingabefeld benötigt eine Steuervariable. Im Beispiel ist das die Variable:

`textvariable = eingabefeld_wert`

Die Steuervariable `eingabefeld_wert` wird im Hauptprogramm erzeugt (Zeile 20) und mit der Eingabe geschrieben (Zeile 23).

Wir erzeugen zusätzlich eine Steuervariable (Zeile 27) für das Ausgabefeld (Zeile 31).

Wir erzeugen eine Schaltfläche (Zeile 35), die eine Funktion aufruft, wenn die Schaltfläche betätigt wird. Im Beispiel ist es die Funktionen:

`command = celsius_in_kelvin`

Diese Funktion wird oben im Programm definiert (Zeile 11 bis 17), damit sie bekannt ist, bevor die Schaltfläche erzeugt wird.

So läuft das Programm ab:

Zeile 23: Das Programm bekommt einen Wert über das Eingabefeld.

Zeile 35: Das Programm erkennt eine Betätigung der Schaltfläche. Aufruf von `celsius_in_kelvin`.

Zeile 13: Die Funktion liest die Steuervariable des Eingabefelds, wandelt von String in Integer und schreibt das Ergebnis in die lokale Variable `celsius`.

Zeile 15: Die Funktion berechnet die lokale Variable `kelvin`.

Zeile 17: Die Funktion schreibt die lokale Variable `kelvin` in die Steuervariable des Ausgabefelds.

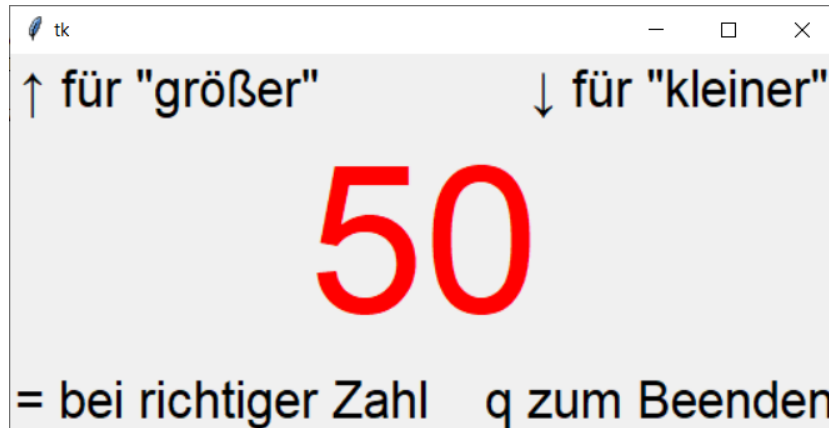
Zeile 31: Das Programm zeigt den berechneten Wert im Ausgabefeld.

Das Widget Schaltfläche benötigt die Angabe einer Funktion, die ausgeführt wird, wenn die Schaltfläche betätigt wird. Beispiel:

command = actionSF1

Computer errät die Zahl

Wir können auch an die Tasten unserer PC-Tastatur eine Funktion binden, die ausgeführt wird, wenn eine Taste betätigt wird. Beispiel dafür ist das Programm "Computer errät die Zahl" in der GUI-Version. So sieht das GUI aus:



Und so sehen die Zeilen 1 bis 37 unseres Programms aus.

```

1  # Computer errät die Zahl - GUI Version
2  # Der Benutzer hilft: Benutzer-Zahl ist größer, kleiner, richtig
3
4  # Modul für die GUI-Erstellung importieren
5  import tkinter as tk
6
7  def rate_zahl():
8      global obere, untere
9      # Steuer-Variable setzen
10     x.set((obere + untere)//2)
11
12 def zeige_kommentar():
13     # Kommentar erzeugen und im Gitter platzieren
14     label4 = tk.Label(root,
15                       text="Zahl erraten!           Versuche = " + str(versuche),
16                       fg="magenta",
17                       font = ("arial", 25))
18     label4.grid(row=3, column=0)
19
20 def onKeyPress(event):
21     global versuche, obere, untere
22     # Tasten auswerten
23     if event.keysym == "Down":
24         versuche += 1
25         obere = max(untere, x.get() - 1)
26         rate_zahl()
27     elif event.keysym == "Up":
28         versuche += 1
29         untere = min(obere, x.get() + 1)
30         rate_zahl()
31     elif event.keysym == "equal":
32         zeige_kommentar()
33     elif event.keysym == "q":
34         root.destroy()
35     else:
36         return
37

```

Wir sprechen zuerst über die Zeilen 38 bis 77 unseres Programms.

```

38 # Konstruktor für das Fenster aufrufen
39 root = tk.Tk()
40
41 # untere und obere Grenze der Zahl
42 untere = 1
43 obere = 100
44
45 # Anzahl der Versuche
46 versuche = 0
47
48 # Bedienungsanleitung-1 erzeugen und im Gitter platzieren
49 label1 = tk.Label(root,
50                   text = "↑ für \"größer\"           ↓ für \"kleiner\"",
51                   font = ("arial", 25))
52 label1.grid(row=0, column=0)
53
54 # Steuer-Variable für die geratene Zahl erzeugen
55 x = tk.IntVar()
56
57 # Steuer-Variable initialisieren
58 rate_zahl()
59
60 # Ausgabe der Zahl erzeugen und im Gitter platzieren
61 label2 = tk.Label(root,
62                   textvariable = x,
63                   fg="red",
64                   font = ("arial", 100))
65 label2.grid(row=1, column=0)
66
67 # Bedienungsanleitung-2 erzeugen und im Gitter platzieren
68 label3 = tk.Label(root,
69                   text ="= bei richtiger Zahl    q zum Beenden",
70                   font = ("arial", 25))
71 label3.grid(row=2, column=0)
72
73 # Auf Tastendruck reagieren, wenn Focus im Fenster ist
74 root.bind("<KeyPress>", onKeyPress)
75
76 # Hauptschleife, damit die GUI angezeigt wird
77 root.mainloop()
    
```

Im Hauptprogramm definieren wir die Variablen für die Grenzen und die Anzahl der Versuche (Zeilen 42 bis 46). Wir schreiben eine Bedienungsanleitung für unser Programm ins Fenster (Zeilen 48 bis 52 und 67 bis 71). Wir erzeugen die Steuervariable für die geratene Zahl und initialisieren sie durch den Aufruf der Funktion `rate_zahl` (Zeilen 54 bis 58).

Neu: Wir binden an die Tasten unserer PC-Tastatur die Funktion `onKeyPress` (Zeile 74). `onKeyPress` wird ausgeführt, wenn eine Taste betätigt wird.

Nun sprechen wir über die Zeilen 1 bis 37 unseres Programms. Dort stehen die Funktionen. Die Funktionen müssen definiert werden, bevor wir in den Zeilen 38 bis 77 auf sie verweisen!

Die Funktion `rate_zahl` berechnet die Zahl, die der Computer als nächstes rät (Zeilen 7 bis 10). Die Funktion `zeige_kommentar` zeigt uns zum Abschluss die Anzahl der Versuche (Zeilen 12 bis 18).

Die Funktion `onKeyPress` wird bei jedem Tastendruck mit dem Eingangswert `event` aufgerufen. Nun können wir mit `event.keysym` die Taste abfragen. `tkinter` erkennt eine Taste an ihrem symbolischen Namen (`keysym`). Wir verwenden "Down", "Up", "equal" und "q". Die If-Abfrage enthält Anweisungen für jede Taste.

Hinweis zu Operatoren: // dividiert und schneidet die Werte nach dem Komma ab.
 % liefert den Rest nach einer Division.

15 Turtle Grafik

Turtle-Grafik wurde in den späten 1960er-Jahren für den Unterricht entwickelt. Ein Stift-tragender Roboter – die Schildkröte – bewegt sich auf der Zeichenebene und hinterlässt dabei eine Spur.

Die Schildkröte (turtle) hat 3 Attribute: Ort, Richtung und Stift.

In Python ist ein Turtle-Modul integriert. Mit wenigen Befehlen zeichnet man ein gefülltes Dreieck:

```
1 # Zeichne ein gefülltes rechtwinkliges Dreieck
2
3 # Modul für die Turtle-Grafik importieren
4 import turtle
5
6 # Bildschirmobjekt erzeugen
7 screen = turtle.Screen()
8
9 # Turtle-Geschwindigkeit setzen
10 turtle.speed(1)
11
12 # Hintergrundfarbe des Bildschirms setzen
13 turtle.bgcolor('black')
14
15 # Stiftfarbe setzen
16 turtle.color('yellow')
17
18 # Strichstärke setzen
19 turtle.width(10)
20
21 # Gefüllte Form starten
22 turtle.begin_fill()
23
24 # Vorwärts 100 Schritte
25 turtle.forward(100)
26
27 # Rechtsherum drehen um 90 Grad
28 turtle.right(90)
29
30 # Vorwärts 100 Schritte
31 turtle.forward(100)
32
33 # Füllfarbe setzen
34 turtle.color('orange')
35
36 # Gefüllte Form beenden
37 turtle.end_fill()
38
39 # Hauptschleife, damit die Turtle-Grafik angezeigt wird
40 screen.mainloop()
```

Turtle macht das Fenster 400 x 300 Schritte groß. Für eine eigene Fenstergröße benutzen wir die Methode `screen.size()`. Beispiel: `screen.size(800, 600)`.

Wenn wir die Methoden in einer Schleife aufrufen, können wir geometrische Figuren erzeugen.

Beispiel:

```
1 # Zeichne ein Quadrat
2
3 # Modul für die Turtle-Grafik importieren
4 from turtle import *
5
6 # Bildschirmobjekt erzeugen
7 screen = Screen()
8
9 # Turtle Geschwindigkeit: langsamste
10 speed(1)
11
12 # Strichstärke setzen
13 width(3)
14
15 # Stiftfarbe setzen
16 color("green")
17
18 # Quadrat
19 for i in range(4):
20     forward(100)
21     left(90)
22
23 # Hauptschleife, damit die Turtle-Grafik angezeigt wird
24 screen.mainloop()
```

Durch den Import in Zeile 4 werden alle Namen im Modul im Programm bekannt gemacht. Deshalb können wir in den folgenden Zeilen die Angabe "turtle" vor dem Namen weglassen.

Zeile 10 speed(1) bewirkt, dass wir der Schildkröte beim Zeichnen zusehen können. Es gilt:

```
'fastest' : 0
'fast'    : 10
'normal'  : 6
'slow'    : 3
'slowest' : 1
```

Turtle kann den Benutzer nach einem Text oder einer Zahl fragen und beides im Fenster ausgeben.

Beispiel:

```
1 # Demonstration von textinput und numinput
2
3 # Modul für die Turtle-Grafik importieren
4 from turtle import *
5
6 # Bildschirmobjekt erzeugen
7 screen = Screen()
8
9 # Strichstärke setzen
10 width(3)
11
12 # Stiftfarbe setzen
13 color("blue")
14
15 # Name abfragen
16 name = screen.textinput("textinput", "Wie heißt du?")
17
18 # Stift abheben, an Position platzieren, aufsetzen
19 up()
20 goto(-200, 0)
21
22 # Name schreiben und Turtle ans Ende bewegen
23 write("Name: " + name + " ", move=True, font=("Arial", 24))
24
25 # Klasse abfragen
26 klasse = screen.numinput("numinput", "In welcher Klasse bist du?", minval=1, maxval=13)
27
28 # In Integer umwandeln
29 klasse = int(klasse)
30
31 # Klasse schreiben und Turtle ans Ende bewegen
32 write("Klasse: " + str(klasse), move=True, font=("Arial", 24))
33
34 # Hauptschleife, damit die Turtle-Grafik angezeigt wird
35 screen.mainloop()
```

Zeile 16 fragt nach einem Text. Zeile 26 fragt nach einer Zahl. numinput() fängt Eingabefehler ab! Wenn der Benutzer eine Zahl kleiner 1 oder größer 13 eingibt, gibt es eine Fehlermeldung und der Benutzer bekommt eine zweite Chance. Ebenso, wenn der Benutzer einen Buchstaben oder ein Sonderzeichen eingibt. Die Zeilen 23 und 32 schreiben ins Fenster. move=True sorgt dafür, dass die Schildkröte die Texte hintereinander in das Fenster schreibt.

Die Turtle-Methoden können mit verschiedenen Namen aufgerufen werden. Beispiele:

```
forward() | fd()
backward() | bk() | back()
right() | rt()
left() | lt()
goto() | setpos() | setposition()
pendown() | pd() | down()
penup() | pu() | up()
pensize() | width()
```

16 Dateien lesen

Wir wollen Daten aus einer Datei lesen. Es ist gut zu wissen, wie unser Arbeitsordner heißt, damit wir eine Datei zum Testen dorthin kopieren können. Wir importieren das Modul `os` und lassen uns den Arbeitsordner mit `os.getcwd()` ausgeben. Anschließend öffnen wir mit dem Befehl `open()` die Datei. Wir lesen die Datei und speichern den Inhalt in der Variablen `mein_text`. Danach schließen wir die Datei.

```
1 # Datei (ANSI Codierung) lesen und ausgeben
2
3 # Modul mit Betriebssystem-Funktion importieren
4 import os
5
6 # Den aktuellen Arbeitsordner ausgeben
7 print("Der Arbeitsordner ist:", os.getcwd())
8
9 # Version mit open() und close()
10 # Datei öffnen
11 datei = open("textdatei.txt")
12 # Datei lesen
13 mein_text = datei.read()
14 # Nicht vergessen: Datei schließen!
15 datei.close()
```

Häufig wird vergessen, die Datei nach dem Lesen oder Schreiben zu schließen. Das kann dazu führen, dass nicht der ganze Inhalt gelesen oder geschrieben wird. Wenn wir den Befehl `with` verwenden, kümmert sich Python um das Schließen der Datei:

```
20 # Version mit dem Befehl with. Dadurch wird die Datei automatisch geschlossen.
21 # Datei öffnen, lesen und schließen
22 with open("textdatei.txt") as datei:
23     mein_text = datei.read()
```

Nun geben wir den Inhalt der Datei `textdatei.txt` aus:

```
25 # Inhalt der Datei ausgeben
26 print(mein_text)
```

Ergebnis:

```
Eine Möhre in der Hand
Zwei Äpfel im Korb
Drei Hühner im Hof
```

Beachte: Die deutschen Umlaute ä, ö, ü werden korrekt dargestellt, weil unser PC mit der für uns passenden Voreinstellung für die Codierung arbeitet. In der IDLE-Shell können wir uns die Voreinstellung anzeigen lassen:

```
>>> import locale
>>> locale.getpreferredencoding()
'cp1252'
```

Codepage `cp1252` bedeutet Western Europe. `cp1252` ist auch unter dem Begriff ANSI bekannt, obwohl es kein ANSI-Standard ist.

Um alle Zeichen aller natürlichen Sprachen (u. a. Griechisch, Arabisch, Chinesisch) darstellen zu können, wurde der Unicode entwickelt. Unicode Zeichen werden meistens in UTF-8 codiert. (Dabei entsprechen die ersten 127 Zeichen der ASCII Codierung.)

Die Datei `textdatei_utf_8.txt` ist UTF-8 codiert. Um diese Datei korrekt zu lesen, müssen wir beim Befehl `open()` die Codierung mit angeben:

```
20 # Version mit dem Befehl with. Dadurch wird die Datei automatisch geschlossen.
21 # Datei öffnen, lesen und schließen
22 with open("textdatei_utf_8.txt", encoding="utf-8") as datei:
23     mein_text = datei.read()
```

Nun wollen wir nicht unser Python-Programm ändern, wenn wir herausfinden wollen, ob eine Datei ANSI-codiert oder UTF-8-codiert ist. Stattdessen wollen wir dem Programm zwei_textcodierungen.py beim Aufruf einen Kommandozeilen-Parameter mitgeben, mit dem wir die Codierung und die Datei auswählen.

So soll der Aufruf unseres Programms aussehen:

```
zwei_textcodierungen.py ansi
- oder -
zwei_textcodierungen.py utf-8
```

Die Funktion parameterlesen liest den Kommandozeilen-Parameter. Wir importieren wir das Modul sys, das uns Zugriff auf den Python-Interpreter gibt. Mit der Liste sys.argv erhalten wir den Kommandozeilenparameter:

```
sys.argv[0] enthält den Namen des Python-Programms
sys.argv[1] enthält den Kommandozeilen-Parameter
6 # Modul mit Zugriff auf den Interpreter importieren
7 import sys
8
9 # Kommandozeilen-Parameter lesen
10 def parameterlesen() -> str:
11     if len(sys.argv) != 2:
12         return("Fehler")
13     else:
14         return(sys.argv[1])
```

Kommandozeilen-Parameter lesen

Zwei Parameter?	
wahr	falsch
den zweiten Parameter zurückgeben	"Fehler" zurückgeben

Im Hauptprogramm rufen wir die Funktion parameterlesen auf und bewerten den Rückgabewert. Abhängig vom Rückgabewert befassen wir uns mit der Datei textdatei.txt oder textdatei_utf_8.txt.

Zwei Textcodierungen

Den aktuellen Arbeitsordner ausgeben			
Kommandozeilen-Parameter lesen			
Parameter?			
"ansi"	"utf-8" oder "utf_8"	"Fehler"	sonst
name = "textdatei.txt" ok = True	name = "textdatei_utf_8.txt" ok = True	"Parameter fehlt" ok = False	"Parameter unbekannt" ok = False
ok?			
wahr		falsch	
name ausgeben Datei name öffnen, lesen, schließen		nichts tun	
ok?			
wahr		falsch	
Inhalt der Datei ausgeben		nichts tun	

Wenn die Datei (z. B. info.txt) im aktuellen Arbeitsordner liegt, wird sie vom Befehl open() gefunden. Wenn info.txt in einem anderen Ordner liegt, müssen wir im Befehl open() den Dateipfad angeben.

Beispiel:

Der Arbeitsordner ist: c:\Users\marti\Documents\Python\Datei_lesen_schreiben\Tag_8

open() mit Dateipfad und Datei:

```
36| with open("c:/Users/marti/Documents/Python/info.txt", encoding=parameter) as datei:
37|     mein_text = datei.read()
```

Wichtig: Wir trennen die Ordner und Unterordner durch einen Schrägstrich /.

Wenn wir den Backslash verwenden wollen, müssen wir zweimal Backslash \\ angegeben, weil der Backslash ein Steuerzeichen ist.

17 Dateien schreiben

Nun wollen wir nicht nur Daten aus einer Datei lesen, sondern auch Daten in eine Datei schreiben. Dazu müssen wir im Befehl open() den Modus "w" angeben (w – write).

Wir wollen zum Beispiel eine Datei lesen, Text ersetzen und eine Datei schreiben.

Schritt 1: Datei öffnen, lesen und schließen. Wir geben im Befehl open() den Modus "r" an (r – read). Das könnten wir auch weglassen, weil "r" die Voreinstellung ist.

```
31| with open("bericht1.txt", mode="r", encoding=parameter) as datei:
32|     der_ganze_text = datei.read()
```

Schritt 2: Nun steht der ganze Text in der String-Variablen der_ganze_text. Auf diesen String können wir alle Methoden der Klasse str anwenden. Wir verwenden die Methode replace(), um Text zu ersetzen.

```
40| der_ganze_text = der_ganze_text.replace("Luftschiffe", "Zeppeline")
```

Schritt 3: Datei öffnen, schreiben und schließen. Wir geben im Befehl open() den Modus "w" an.

```
48| with open("bericht2.txt", mode="w", encoding=parameter) as datei:
49|     datei.write(der_ganze_text)
```

Manchmal liegen unsere Daten in einer EXCEL-Datei vor. Diese Daten können wir im Python-Programm einlesen, nachdem wir die Daten der EXCEL-Datei im CSV-Format abgespeichert haben. In EXCEL wählen wir:

"Speichern unter" "Dateityp: CSF UTF-8 (durch Trennzeichen getrennt) (*.csv)"

Im Python-Programm importieren wir das Modul für die CSV Datei:

```
import csv
```

Dann öffnen wir die CSV-Datei und lesen die Dateien mit der Methode csv.reader(). Anschließend schreiben wir jede Zeile in eine Liste.

```
39| # Vokabelliste initialisieren
40| meine_liste = []
41|
42| # CSV-Datei öffnen, lesen, schließen
43| with open("idioms.csv", encoding="utf-8") as csvdatei:
44|     csv_reader_object = csv.reader(csvdatei, delimiter=";")
45|     # Jede Zeile in die Liste schreiben
46|     for row in csv_reader_object:
47|         meine_liste.append(row)
```

Wir erhalten ein zweidimensionales Array, das wir in Python weiterverarbeiten können.

18 Objektorientierte Programmierung (OOP) – Klassen

Bisher haben wir entweder Daten geschrieben oder Aufgaben mit Funktionen erledigt. Bei der objektorientierten Programmierung (OOP) verknüpfen wir Daten und Methoden. Wir beschreiben ein Objekt (Ding). Beispiel "Katze":

Eigenschaften: Name, Farbe, Alter

Methoden: miauen, schlafen

Begriffe der OOP

Klassen (der Bauplan)

Objekt (aus Klassen erstellt)

Instanz = Objekt

Eigenschaften (oder Attribute)

Methoden (Funktionen, was das Objekt tun kann)

Vererbung (Eltern-Klasse, Kind-Klasse)

Wir erstellen eine Katzenklasse mit Eigenschaften:

```

1  # Objektorientierte Programmierung
2
3  # Klassendefinition
4  class BauplanKatzenKlasse():
5      """ Klasse für das Erstellen von Katzen
6      Hilfetext ideal bei mehreren Programmierern in
7      einem Projekt oder bei schlechtem Gedächtnis """
8
9      # Methoden der Klasse
10     def __init__(self, rufname, farbe, alter):
11         self.rufname = rufname
12         self.farbe = farbe
13         self.alter = alter
14
15     # Instanz Sammy erstellen
16     katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
17     print(katze_sammy.alter)
18
    
```

Es ist üblich, den Klassennamen groß zu schreiben. Wörter werden durch Großbuchstaben im Klassennamen getrennt. Mit drei Anführungszeichen umschließen wir den Hilfetext, der über mehrere Zeilen geht (""" ... """).

Wenn wir das Programm mit der Klasse in der IDLE-Shell starten, passiert gar nichts. Aber die Klasse ist jetzt definiert. Wir lassen uns den Hilfetext anzeigen:

`help(BauplanKatzenKlasse)`. Ausgabe:

Help on class BauplanKatzenKlasse in module __main__:

```

class BauplanKatzenKlasse(builtins.object)
|   BauplanKatzenKlasse(rufname, farbe, alter)
|
|   Klasse für das Erstellen von Katzen
|   Hilfetext ideal bei mehreren Programmierern in
|   einem Projekt oder bei schlechtem Gedächtnis
|
|   Methods defined here:
|
|   __init__(self, rufname, farbe, alter)
|       Initialize self.  See help(type(self)) for accurate signature.
|
|   tut_miauen(self, anzahl=1)
|
|   tut_schlafen(self, dauer)
    
```


Die Methode `__init__(self, ...)` führt die Eigenschaften der Klasse ein. Die Parameterliste beginnt dabei immer mit `self`. Was bedeutet `self`?

Bedeutung von `self` (© <https://www.python-lernen.de/>)

Objekt erstellen (außerhalb der Klasse)

```
katze_sammy = BauplanKatzenKlasse(3)
```

Übergabe in der Klasse

```
BauplanKatzenKlasse().__init__(katze_sammy, 3)
```

```
def __init__(self, alter):
    self.alter = alter
```

Abruf außerhalb der Klassen:

```
print(katze_sammy.alter)
```

Wenn wir die Instanz der Klasse erstellen, tritt `katze_sammy` an die Stelle von `self`.

"Sammy", "orange", 3 treten an die Stelle von `rufname`, `farbe`, `alter`.

Beim Abruf außerhalb der Klasse tritt `katze_sammy` an die Stelle von `self` und es wird `katze_sammy.alter` zugegriffen:

```
print(katze_sammy.alter)
```

Bisher hat die Katzenklasse nur Eigenschaften. Nun ergänzen wir die Methoden:

```
1 # Objektorientierte Programmierung
2
3 # Klassendefinition
4 class BauplanKatzenKlasse():
5     """ Klasse für das Erstellen von Katzen
6     Hilfetext ideal bei mehreren Programmierern in
7     einem Projekt oder bei schlechtem Gedächtnis """
8
9     # Methoden der Klasse
10    def __init__(self, rufname, farbe, alter):
11        self.rufname = rufname
12        self.farbe = farbe
13        self.alter = alter
14        # Dauer aufaddieren
15        self.schlafdauer = 0
16
17    def tut_miauen(self, anzahl = 1):
18        print(anzahl * "miau ")
19
20    def tut_schlafen(self, dauer):
21        print(self.rufname, "schläft jetzt", dauer, "Minuten ")
22        self.schlafdauer += dauer
23        print(self.rufname, "Schlafdauer insgesamt:", self.schlafdauer, "Minuten ")
24
25    # Instanz Sammy erstellen
26    katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
27    print(katze_sammy.alter)
28
```

Außerdem ergänzen wir in der Methode `__init__` eine Eigenschaft, mit deren Hilfe wir die Schlafdauer aufaddieren:

```
self.schlafdauer = 0
```

Wenn wir jetzt eine zweite Instanz der Katzenklasse erstellen und die Methoden aufrufen, sehen wir, dass die Schlafdauer für die Instanzen `katze_sammy` und `katze_soni` getrennt aufaddiert wird.

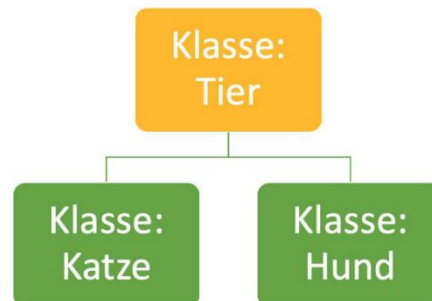
```
31 # Instanz Soni erstellen
32 katze_soni = BauplanKatzenKlasse("Soni", "getigert", 2)
33
34 # Methoden von Katze Sammy aufrufen
35 katze_sammy.tut_miauen(3)
36 katze_sammy.tut_schlafen(3)
37 katze_sammy.tut_schlafen(6)
38 katze_sammy.tut_miauen(5)
39 katze_sammy.tut_schlafen(10)
40
41 # Methode von Katze Soni aufrufen
42 katze_soni.tut_schlafen(5)
43
```

```
miau miau miau
Sammy schläft jetzt 3 Minuten
Sammy Schlafdauer insgesamt: 3 Minuten
Sammy schläft jetzt 6 Minuten
Sammy Schlafdauer insgesamt: 9 Minuten
miau miau miau miau miau
Sammy schläft jetzt 10 Minuten
Sammy Schlafdauer insgesamt: 19 Minuten
Soni schläft jetzt 5 Minuten
Soni Schlafdauer insgesamt: 5 Minuten
```

19 OOP – Vererbung bei Klassen

Vererbung bei Klassen: Die Kind-Klasse übernimmt alle Eigenschaften und Methoden der Eltern-Klasse. Der Überbegriff zur Katze ist Tier. Genau genommen handelt es sich um ein Säugetier. Wir erstellen die Klasse Tier, die Katze und Hund beschreiben soll.

Unsere Klassen mit Hierarchie (© <https://www.python-lernen.de/>)



Wir erstellen die Klasse Tier, die alle Eigenschaften der unserer bisherigen Klasse Katze erhält. Dadurch wird die Klasse Katze bereinigt – doppelt ist unnötig.

```

1 # Vererbung in Python
2
3 # Eltern-Klasse
4 class Tier():
5     """ Klasse für das Erstellen von Säugetieren """
6
7     def __init__(self, rufname, farbe, alter):
8         self.rufname = rufname
9         self.farbe = farbe
10        self.alter = alter
11        self.schlafdauer = 0
12
13    def tut_schlafen(self, dauer):
14        print(self.rufname, "schläft jetzt", dauer, "Minuten ")
15        self.schlafdauer += dauer
16        print(self.rufname, "Schlafdauer insgesamt:", self.schlafdauer, "Minuten")
17
18 # Kind-Klasse für Katzen
19 class BauplanKatzenKlasse(Tier):
20     """ Klasse für das Erstellen von Katzen """
21
22     def __init__(self, rufname, farbe, alter):
23         """ Initialisieren über Eltern-Klasse """
24         super().__init__(rufname, farbe, alter)
25
26 # Instanz Sammy erstellen
27 katze_sammy = BauplanKatzenKlasse("Sammy", "orange", 3)
28 print(katze_sammy.farbe)
29
    
```

Zeile 19: Wir aktivieren die Vererbung zwischen Eltern-Klasse und Kind-Klasse, indem wir hinter dem Namen der Klasse die Eltern-Klasse in Klammern notieren.

Zeile 22 - 24: Die `__init__` Methode wird weiterhin benötigt. Die Eigenschaften der Eltern-Klasse werden über `super()` von der Eltern-Klasse übernommen.

Nun ist die Katzenklasse schlanker geworden, wir haben aber noch keinen Code gespart. Das passiert erst, wenn wir eine zweite Kind-Klasse anlegen, also die Klasse Hund.

```

29 # Kind-Klasse für Hunde
30 class Hund(Tier):
31     """ Klasse für das Erstellen von Hunden """
32
33     def __init__(self, rufname, farbe, alter):
34         """ Initialisieren über Eltern-Klasse """
35         super().__init__(rufname, farbe, alter)
36
37 # Instanz Bello erstellen
38 hund_bello = Hund("Bello", "braun", 5)
39 print(hund_bello.farbe)
40
    
```

Nun ergänzen wir Methoden in der Eltern-Klasse.

```

1 | # Vererbung in Python|
2 |
3 | # Eltern-Klasse
4 | class Tier():
5 |     """ Klasse für das Erstellen von Säugetieren """
6 |
7 |     def __init__(self, rufname, farbe, alter):
8 |         self.rufname = rufname
9 |         self.farbe = farbe
10 |        self.alter = alter
11 |        self.schlafdauer = 0
12 |
13 |    def tut_schlafen(self, dauer):
14 |        print(self.rufname, "schläft jetzt", dauer, "Minuten ")
15 |        self.schlafdauer += dauer
16 |        print(self.rufname, "Schlafdauer insgesamt:", self.schlafdauer, "Minuten")
17 |
18 |    def tut_reden(self, anzahl = 1):
19 |        print(self.rufname, "sagt: ", anzahl * "miau ")
20 |

```

Weder in der Klasse Katze noch in der Klasse Hund müssen wir etwas tun. Beide erben automatisch von der Eltern-Klasse "Tier". Wenn wir die Methoden aufrufen, können wir mit der Ausgabe der Methode tut_schlafen() zufrieden sein.

Aber die Ausgabe der Methode tut_reden() passt für den Hund nicht.

```

45 | # Methoden aufrufen
46 | hund_bello.tut_schlafen(4)
47 | katze_sammy.tut_schlafen(5)
48 | hund_bello.tut_schlafen(2)
49 | katze_sammy.tut_reden(1)
50 | hund_bello.tut_reden(3)
51 |
orange
braun
Bello schläft jetzt 4 Minuten
Bello Schlafdauer insgesamt: 4 Minuten
Sammy schläft jetzt 5 Minuten
Sammy Schlafdauer insgesamt: 5 Minuten
Bello schläft jetzt 2 Minuten
Bello Schlafdauer insgesamt: 6 Minuten
Sammy sagt: miau
Bello sagt: miau miau miau

```

Überschreiben von Methoden: Wir können in der Kind-Klasse Hund die Methode tut_reden() der Eltern-Klasse überschreiben. In der Klasse Hund erstellen wir eine Methode mit demselben Namen.

```

29 | # Kind-Klasse für Hunde
30 | class Hund(Tier):
31 |     """ Klasse für das Erstellen von Hunden """
32 |
33 |     def __init__(self, rufname, farbe, alter):
34 |         """ Initialisieren über Eltern-Klasse """
35 |         super().__init__(rufname, farbe, alter)
36 |
37 |     def tut_reden(self, anzahl = 1):
38 |         """ Überschreiben der Methode der Eltern-Klasse """
39 |         print(self.rufname, "sagt: ", anzahl * "WAU ")
40 |
orange
braun
Bello schläft jetzt 4 Minuten
Bello Schlafdauer insgesamt: 4 Minuten
Sammy schläft jetzt 5 Minuten
Sammy Schlafdauer insgesamt: 5 Minuten
Bello schläft jetzt 2 Minuten
Bello Schlafdauer insgesamt: 6 Minuten
Sammy sagt: miau
Bello sagt: WAU WAU WAU

```

In der Kind-Klasse können wir auch eine zusätzliche Eigenschaft einführen und eine zusätzliche Methode erstellen.

20 Bibliothek pygame

Die Pygame Bibliothek ist wohl die bekannteste Bibliothek zum Programmieren von Spielen. Sie enthält zahlreiche Module mit Funktionen zum Zeichnen, zum Abspielen von Tönen, zur Verarbeitung von Eingaben mit der Tastatur und der Maus usw. Bei unserem Spiel geht es um ein Auto, das wir seitwärts bewegen können, um dem entgegenkommenden Auto auszuweichen. Das Programm [Some_Shapes.py](#) (shape – Form) zeigt uns die Grundlagen.

Zeile 1: Wir importieren die Module.

Zeile 2: Diese Zeile erlaubt uns, Funktionen direkt aufzurufen, ohne pygame.locals davor zu setzen.

Zeile 5: Zu Beginn wird pygame.init() aufgerufen.

Der (unendliche) Game Loop geht von Zeile 35 bis 42. (In diesem Programm benötigen wir keinen Game Loop. Wir können ihn aber hier anschauen, bevor wir ihn später im Spiel verwenden.)

Zeile 36: Änderungen im Programm werden erst dann wirksam, wenn display.update() gerufen wird.

Zeile 37: Ereignis (event) holen

Zeile 38: Wenn Ereignis QUIT (Beenden)

Zeile 39: quit() setzt alle pygame module zurück

Zeile 40: exit() beendet das Python Script

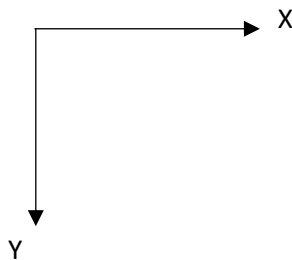
Wir initialisieren ein Fenster für die Anzeige.

Zeile 19: set_mode() erstellt eine Anzeige in der gewünschten Größe.

Zeile 20: fill() füllt das Fenster mit der gewählten Farbe.

Zeile 21: set_caption() setzt den Fenstertitel

In Pygame erstellen wir Objekte mit Zeichenfunktionen. Der Ursprung unseres Koordinatensystems ist in der oberen linken Ecke. X-Werte werden von links nach rechts größer. Y-Werte werden von oben nach unten größer.



Wir zeichnen ein paar Formen:

Zeile 24: line() zeichnet eine Linie vom Startpunkt zum Endpunkt

Zeile 27: circle() zeichnet einen Kreis um den Mittelpunkt mit dem angegebenen Radius

Zeile 29: rect() zeichnet ein Rechteck. Die Maße werden mit einem Tupel angegeben:

1. Die X-Koordinate der oberen linken Ecke des Rechtecks
2. Die Y-Koordinate der oberen linken Ecke des Rechtecks
3. Die Breite des Rechtecks in Pixels.
4. Die Höhe des Rechtecks in Pixels.

Computer sind sehr schnell. Sie können eine Schleife Millionen Male in weniger als 1 Sekunde durchlaufen. Das ist für Menschen zu schnell. Zum Vergleich: Filme werden mit 24 Bildern (Frames) pro Sekunde abgespielt. Bei einem langsameren Tempo ruckelt es. Bei einem Tempo schneller als 100 Bildern pro Sekunden können wir den Objekten nicht mehr folgen.

Wir beschränken also das Tempo für das Spiel.

Zeile 8: FPS = 30 (Frames per Second)

Zeile 9: Clock() erstellt ein Objekt zur Zeiterfassung

Zeile 42: tick() wartet, damit das Spiel nicht schneller läuft als FPS

Some_Shapes.py erstellt und füllt ein Fenster und zeichnet Formen hinein.

Das Programm zeigt die Grundlagen – es ist noch kein Spiel.

[Traffic_Game_Beta.py](#) ist die Beta-Version unseres Spiels. Das Spiel ist nicht perfekt, aber der Anfang ist gemacht. Es geht es um ein Auto, das wir seitwärts bewegen können, um dem entgegenkommenden Auto auszuweichen. Wir programmieren das Spiel objektorientiert, also mit Klassen. Dadurch können wir Code-Abschnitte mehrfach im Programm verwenden.

Wir erstellen die Klasse für das Auto, das wir seitwärts bewegen können (hier: Player).

Zeile 46: Die Klasse Player erbt alle Eigenschaften und Methoden von der Eltern-Klasse `pygame.sprite.Sprite`. (Ein Sprite ist ein Grafikobjekt, das vor dem Hintergrund eingeblendet wird.)

Zeile 47: Wir erstellen die Methode `__init__()`

Zeile 48: `super()` holt alle Eigenschaften von der Eltern-Klasse ab.

Zeile 49: `load()` lädt das Bild mit dem Auto.

Zeile 50: `rect()` erzeugt automatisch ein Rechteck in der Größe unseres Bildes.

Zeile 51: `rect.center` bestimmt den Startpunkt des Rechtecks. Später werden wir das Bild in dieses Rechteck zeichnen. `rect` und `image` müssen immer an derselben Stelle sein!

Zeile 53: `update()` steuert die Bewegung des Players. Die Funktion prüft, ob eine Taste gedrückt ist. Wenn ja, bewegt `move_ip()` den Player in die gewünschte Richtung.

Zeile 56 und 59: Die if-Abfragen stellen sicher, dass der Player das Fenster nicht verlässt.

Zeile 62: `draw()` zeichnet den Player

Zeile 63: `blit()` zeichnet das Bild in das Rechteck. (blit kommt von BLT – Block Transfer)

Nun erstellen wir die Klasse für das Auto, das entgegenkommt (hier: Enemy). Die Klasse ist ähnlich der Klasse Player. Unterschiede:

Zeile 35: Der Startpunkt des Rechtecks ist zufällig. (Deshalb muss Player ausweichen.)

Zeile 37: `move()` steuert die Bewegung des Enemys.

Zeile 38: `move_ip` bewegt den Enemy.

Zeile 39: prüft, ob Enemy am unteren Rand angekommen ist.

Zeile 40 und 41: Wenn ja, zurück an eine zufällige Position am oberen Rand.

Zeile 66 und 67 erstellen je eine Instanz von Player und Enemy.

Der (unendliche) Game Loop geht von Zeile 70 bis 84.

Zeile 76: bewegt den Player

Zeile 77: bewegt den Enemy

Zeile 79: Das Fenster wird neu gefüllt

Zeile 80: zeichnet den Player

Zeile 81: zeichnet den Enemy

Zeile 83: aktualisiert das Fenster

Zeile 84: tick() wartet, damit das Spiel nicht schneller läuft als FPS (hier: 60)

Wir starten das Spiel `Traffic_Game_Beta.py`. Mit den Pfeil-Tasten lassen wir unser Auto (Player) ausweichen. Wenn wir mit dem entgegenkommenden Auto (Enemy) zusammenstoßen, passiert nichts.

Es fehlt also:

- Das Entdecken des Zusammenstoßes
- Die Darstellung von "Game Over"

Das gibt es in der Version [Traffic_Game.py](#). Hier die Kommentare zu den geänderten Zeilen.

Zeile 23: Die Variable `SPEED` setzt die Geschwindigkeit. Diese wird im Spiel erhöht.

Zeile 24: Die Variable `SCORE` zählt, wie oft der Player dem Enemy erfolgreich ausgewichen ist.

Zeile 31: lädt den Hintergrund "Animated Street".

Zeile 48: Der Enemy bewegt sich mit der Geschwindigkeit `SPEED`.

Zeile 50: Hier wird `SCORE` erhöht.

Zeile 62 bis 70: Die Funktion wurde umbenannt. Sie heißt jetzt `move()`. Sonst keine Änderung.

Zeile 71: Die Funktion `draw()` wird nicht mehr gebraucht.

Zeile 77 bis 82: Wir erzeugen zwei Gruppen: Eine für Enemy und eine für alle Sprites.

Zeile 84 bis 86: Wir erzeugen ein neues Benutzer-Ereignis.

Zeile 85: Das Benutzer-Ereignis `INC_SPEED` bekommt eine eindeutige Kennung

Zeile 86: `INC_SPEED` wird alle 1000 Millisekunden – also jede Sekunde – aufgerufen.

Zeile 93 bis 94: Beim Ereignis `INC_SPEED` wird `SPEED` um 0,5 erhöht. Ausweichen wird schwieriger.

Zeile 100: Zeichnet den Hintergrund "Animated Street"

Zeile 101 bis 102: zeigt den `SCORE` an.

Zeile 104 bis 107: Bewegt alle Sprites und zeichnet sie neu.

Zeile 110 bis 123: Hier wird der Zusammenstoß zwischen Player und Enemy behandelt.

Zeile 111: `spritecollideany()` prüft, ob der Player mit irgendeinem Enemy zusammengestoßen ist.

(In unserem Spiel gibt es nur 1 Enemy. Es wären aber auch 1000 Enemy Sprites möglich.)

Zeile 112: Spielt "crash.wav" ab.

Zeile 115: Füllt das Fenster mit Rot.

Zeile 119 und 120: Räumt die Sprites auf.

Zeile 122 und 123: Beendet das Spiel.

Zeile 125: aktualisiert das Fenster beim normalen Spielablauf (kein Zusammenstoß).

Ende

Quellen

Woher	Was
https://docs.python.org/3/	Sprache
https://www.python-lernen.de/	Kursinhalt, Sprache
https://www.deinprogramm.de/	Konstruktionsanleitung, Merksätze, Aufgaben
https://numpy.org/doc/stable/user/index.html	Bearbeitung von Arrays
https://www.labri.fr/perso/nrougier/from-python-to-numpy/#building-a-maze	Building a maze
https://de.wikipedia.org/wiki/Endlicher_Automat	Zustandsmaschine
https://github.com/hobbyelektroniker/StateMachine	Ampel
https://www.mintgruen.tu-berlin.de/robotikWiki/doku.php?id=techniken:zustandsautomaten	Saugroboter
http://micropython.org/	Micropython Software und Dokumentation
https://github.com/jrullan/micropython_statemachine	Bibliothek statemachine
https://github.com/jrullan/micropython_neotimer	Bibliothek neotimer
https://tkdocs.com/shipman/	GUI mit tkinter
https://www.tcl.tk/man/tcl8.4/TkCmd/keysyms.html	Liste der Tasten, die tkinter.Tk erkennt
http://www.coding4you.at/python/	Turtle Aufgaben
https://gist.github.com/wynand1004/	Snake Game
https://de.wikipedia.org/wiki/Liste_der_Unicod_ebl%C3%B6cke	Liste der Unicode-Blöcke
https://www.pygame.org/docs/genindex.html	Liste von Funktionen, Klassen, Methoden in der pygame Bibliothek
https://coderslegacy.com/python/python-pygame-tutorial/	Traffic Game
https://stackoverflow.com/questions	Fragen und Antworten zu Python
https://pypi.org/	Python Package Index (PyPi). Dort liegt Software, die von der Python Community entwickelt wurde, und auf dem eigenen PC installiert werden kann (matplotlib, pygame, ...).
https://pip.pypa.io/en/stable/	Mit dem Paketmanager pip werden Pakete von PyPi zur Installation abgerufen.
RRZN Handbuch (2012), Python – Grundlagen, fortgeschrittene Programmierung und Praxis	Sprache
Felleisen et al. (2013), Realm of Racket, No Starch Press, San Francisco	Spiel
Engelmann u. a. (2017), Duden Informatik S I, Cornelsen Verlag, Berlin	Automaten und Algorithmen