Python Grundkurs

- Einführungsmaterial für Anfänger –

➔ Dieses Material ist ein Projekt in Entwicklung – Rückmeldungen sind erwünscht!

07.2026 (Sch)

Python Grundkurs

Inhaltsverzeichnis:

07.2026 (Sch)

Python Programmierung (Grundlagen)

AB-01: Textausgabe

Textausgabe mit „print“:

a) Um in der Konsole (REPL) Text auszugeben, brauchen wir den Befehl „**print**“. Der auszugebende Text muss dabei entweder in doppelte (“) oder einfache (‘) Anführungszeichen gesetzt werden. Beide Varianten erzeugen einen String (Text).

```
1 print("Hallo Welt")
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Hallo Welt
```

b) Will man **mehrzeiligen Text** ausgeben, kann man mit „\n“ eine neue Zeile beginnen:

```
1 print('Hallo Welt!\n**********')
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Hallo Welt!
  **********
```

c) Man kann auch den **Inhalt einer Variablen** ausgeben, die man zuvor definiert hat:

```
1 text="Hallo Welt!"
2 print(text)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Hallo Welt!
```

d) Mit **formatierten Strings** kann man Variablen und Text kombinieren. Damit das allerdings funktioniert, muss dem String ein „**f**“ vorangestellt werden.

```
1 name="Peter"
2 print(f"Hallo {name}!")
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Hallo Peter!
```

e) Man kann auch verschiedene Datentypen in der Ausgabe kombinieren. Hinter der Raute (#) stehen Kommentare. Sie werden vom Interpreter nicht beachtet.

```
1 name="Peter" #String (Text)
2 alter=16 #Integer (Zahl)
3 print(name, alter)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Peter 16
```

f) Mit dem Operator „+“ können Strings und String-Variablen auch kombiniert werden.

```
1 vorname="Peter"
2 nachname="Lustig"
3 name=vorname+" "+nachname
4 print(name)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Peter Lustig
```

Python Programmierung (Grundlagen)

AB-02: Mathematisch Operationen

Einfaches Rechnen:

a) In der Konsole können **Rechenaufgaben** direkt ausgeführt werden:

```
>>> 4+5
9
>>> 4/5
0.8
```

b) Dabei kann auch auf zuvor gespeicherte **Variablen** zurückgegriffen werden:

```
>>> a=4
>>> b=5
>>> a*b
20
```

c) Es ist auch möglich **Vergleichsoperationen** durchzuführen. Die Abfrage, ob a=b ist, liefert die Rückgabe „False“, da unwahr.

```
>>> a=4
>>> b=5
>>> a==b
False
```

d) Ganze **Zahlen** werden als **integer, Kommazahlen** als **float** gespeichert. Bei **Divisionen** sollte mit **float** gearbeitet werden. **Wichtig:** Kommazahlen mit „.“ schreiben**!**

```
>>> a=4
>>> type(a)
<class 'int'>
>>> b=5.5
>>> type(b)
<class 'float'>
```

e) Man kann auch mit „//“ **Divisionen ohne Rest** durchführen. Der Rest wird hier einfach weggelassen, es wird nicht gerundet.

```
>>> 7/6
1.1666666666666667
>>> 7//6
1
```

f) Die klassische **Rundung** ist auch möglich. Das zweite Argument steht für die Anzahl an Nachkommastellen:

```
>>> round(3.1415)
3
>>> round(3.1415,1)
3.1
>>> round(3.1415,2)
3.14
```

07.2026 (Sch)

Python Programmierung (Grundlagen)

AB-03: Operatoren

Vergleichsoperatoren:

a) Für den Vergleich von Variablen oder Bedingungen gibt es Vergleichsoperatoren:

```
>>> 5==3
False
>>> 5!=3
True
>>> 5<3
False
>>> 5>3
True
>>> 5<=3
False
>>> 5>=3
True
```

==	:	Gleich
!=	:	Ungleich
<	:	Kleiner als
>	:	Größer als
<=	:	Kleiner oder gleich
>=	:	Größer oder gleich

Der Vergleichsoperationen liefern einen booleschen Wert zurück (boolean). Der Datentyp **bool** kann nur die Zustände **True** oder **False** annehmen (großgeschrieben!).

Logische Operatoren:

b)

```
>>> x=5
>>> (x>0) and (x<10)
True
>>> (x<5) or (x>10)
False
>>> not(x>0) and (x<10)
False
>>> (x>4) ^ (x<10)
False
```

and	:	True falls beide Aussagen wahr.
or	:	True falls eine oder zwei Aussagen wahr.
not	:	Invertiert Ergebnis (True→False).
xor	:	True, wenn genau eine Aussage wahr.

→ Die Rückmeldung erfolgt als boolescher Zustand.

Zuweisungsoperatoren (Auswahl):

c)

Operator		Beispiel	Bedeutung
=	→	x=5	(x=5)
+=	→	x+=5	(x=x+5)
-=	→	x-=5	(x=x-5)
*=	→	x*=5	(x=x*5)
/=	→	x/=5	(x=x/5)
//=	→	x//=5	(x=x//5)

➔ Diese Operatoren ermöglichen in der Praxis eine **verkürzte Darstellung**.

07.2026 (Sch)

Python Programmierung (Grundlagen)

AB-04: Benutzereingaben und Typenumwandlungen

Eingaben auswerten:

a) Häufig tritt der Fall auf, dass ein Benutzer dazu aufgefordert wird etwas einzugeben. Diese Eingabe wird über **input** abgefangen und in einer Variablen gespeichert.

```
1 name=input("Gib deinen Namen ein:")
2 print(f"Hallo {name}!")
```

→

```
>>> %Run -c $EDITOR_CONTENT
 Gib deinen Namen ein:Peter
 Hallo Peter!
```

b) Es können auch mehrere Werte abgefragt werden. Mit **split** werden sie getrennt.

```
1 vorname, nachname = input("Vor- und Nachname: ").split()
2 print("Hallo " + vorname, nachname + " !")
```

→

```
>>> %Run -c $EDITOR_CONTENT
Vor- und Nachname: Peter Lustig
Hallo Peter Lustig !
```

c) Es kann auch gezielt ein Datentyp eingelesen werden. Die Prüfung zeigt, dass „alter“ wie gewünscht ein **integer** (Zahl) ist. Wäre man wie in a) vorgegangen, hätte man die Zahl als **string** abgespeichert, da das hier der Standard ist.

```
1 alter=int(input("Gib dein Alter ein:"))
2 print(f"Du bist {alter} Jahre alt!")
```

→

```
>>> %Run -c $EDITOR_CONTENT
 Gib dein Alter ein:40
 Du bist 40 Jahre alt!
>>> type(alter)
<class 'int'>
```

➔ **Vorsicht**: Bei Eingabe von Leerzeichen / Buchstaben kommt es zu einem Fehler!

Typenumwandlungen:

Häufig wird es nötig verschiedene Datentypen umzuwandeln. Eine kleine Übersicht:

Zahl in Text:

```
1 text=str(18)
2 print(text, type(text))
```

```
>>> %Run -c $EDITOR_CONTENT
  18 <class 'str'>
```

Text in Zahl:

```
1 text=int("18")
2 print(text, type(text))
```

```
>>> %Run -c $EDITOR_CONTENT
  18 <class 'int'>
```

Text in Kommazahl:

```
1 text=float("18")
2 print(text, type(text))
```

```
>>> %Run -c $EDITOR_CONTENT
  18.0 <class 'float'>
```

Zahl in True/False:

```
1 text=bool(1)
2 print(text, type(text))
```

```
>>> %Run -c $EDITOR_CONTENT
  True <class 'bool'>
```

Python Programmierung (Grundlagen)

AB-05: Kontrollstrukturen

Bedingungen und Verzweigungen:

a) Für Reaktionen auf unterschiedliche Eingaben brauchen wir **if**, **else** und **elif** (else if).

```
1  alter=int(input("Gib dein Alter ein:"))
2  if alter >= 18:
3      print("Du bist volljährig!")
```

```
>>> %Run -c $EDITOR_CONTENT

  Gib dein Alter ein:20
  Du bist volljährig!
```

Weil die **Bedingung zutrifft** (20>=18) wird die **folgende eingerückte Zeile** ausgeführt.
Trifft die Bedingung nicht zu, z.B. bei der Eingabe von 10, **wird die Zeile übersprungen**.

b) Hier wird auch mit einer **Ausgabe** reagiert, wenn die **Bedingung nicht zutrifft**.

```
1  alter=int(input("Gib dein Alter ein:"))
2  if alter >= 18:
3      print("Du bist volljährig!")
4  else:
5      print("Du bist minderjährig!")
```

```
>>> %Run -c $EDITOR_CONTENT

  Gib dein Alter ein:16
  Du bist minderjährig!
```

c) Mit **elif** (else if) wird die Liste bis zu einer **passenden Bedingung** abgearbeitet.

```
1  alter=int(input("Gib dein Alter ein:"))
2  if alter >= 18:
3      print("Du bist volljährig!")
4  elif alter >= 13:
5      print("Du bist ein Teenager!")
6  else:
7      print("Du bist ein Kind!")
```

```
>>> %Run -c $EDITOR_CONTENT

  Gib dein Alter ein:14
  Du bist ein Teenager!
```

```
>>> %Run -c $EDITOR_CONTENT

  Gib dein Alter ein:7
  Du bist ein Kind!
```

d)

Auf diese Weise können **verschachtelte Programme** erstellt werden:

```
1  print("Führerscheinchecker\n********************")
2  alter = int(input("Gib dein Alter ein:"))
3  if alter >= 18:
4      führerschein = input("Hast du einen Führerschein? (j/n):")
5      if führerschein == "j":
6          print("Du darfst Auto fahren!")
7      else:
8          print("Du brauchst noch einen Führerschein!")
9  else:
10     print("Du bist noch zu jung fürs Auto fahren!")
```

➔ Die richtigen **Einrückungen** sind **entscheidend** für den Ablauf des Programms!

07.2026 (Sch)

Python Programmierung (Grundlagen)

AB-06: for-Schleifen

Wiederholungen mit der for-Schleife:

a) Häufig will man Vorgänge mehrfach durchführen. Dafür ist die **for-Schleife** geeignet:

```
1  print("for-Schleifen Demo\n")
2  anzahl=int(input("Hochzählen bis:"))
3  for i in range(anzahl):
4      print(i)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  for-Schleifen Demo

  Hochzählen bis:3
  0
  1
  2
```

In diesem Fall läuft die Schleife so oft wie der Benutzer eingibt. **Gestartet wird bei 0**.
Die **Variable i** (n) ist eine **Laufzeitvariable** und nur für die Schleifensteuerung nötig.

b) Man kann einer Schleife auch den **Bereich** vorgeben, in dem sie laufen soll:

```
1  for i in range(5,10):
2      print(i)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  5
  6
  7
  8
  9
```

c) Zusätzlich kann man auch die **Schrittgröße** angeben:

```
1  for i in range(0,10,2):
2      print(i)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  0
  2
  4
  6
  8
```

d) Auch der **Inhalt von Listen** kann ausgegeben werden. Listen werden mit [] dargestellt:

```
1  primzahlen=[2,3,5,7,11,13,17]
2  for i in primzahlen:
3      print(i)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  2
  3
  5
  7
  11
  13
  17
```

➔ Listen können z.B. auch andere Datentypen (z.B. strings) enthalten

Python Programmierung (Grundlagen)

AB-07: while-Schleifen

Wiederholungen mit der while-Schleife:

a) Eine **while**-Schleife funktioniert ähnlich wie eine for-Schleife. Sie hat eine Bedingung unter der sie läuft. Ist die Bedingung nicht mehr wahr, wird sie beendet. Im Unterschied zur for-Schleife kann aber **vorzeitig beendet** oder ein **Durchlauf übersprungen** werden.

```
1 liste=[]
2 i=0
3 while i<5:
4     liste.append(input("Namen:"))
5     i+=1
6 print(liste[0],liste[2],liste[4])
```

```
>>> %Run -c $EDITOR_CONTENT

  Namen:Peter
  Namen:Laura
  Namen:Simon
  Namen:Hans
  Namen:Jona
  Peter Simon Jona
```

In diesem Beispiel läuft die Schleife genau 5x. Im Gegensatz zur for-Schleife müssen wir manuell in jedem Durchlauf die **Laufzeitvariable i** weiter hochzählen. Die Funktion **append** fügt einer Liste Einträge hinzu.

→ Beachte: Listeneinträge beginnen immer bei **0**!

b) Eine besondere Form ist die **while True** – Schleife. Die Bedingung ist hier **immer wahr**, daher läuft sie immer weiter. Sie wird nur durch **break** unterbrochen. Diese Schleife ist besonders zur **Prüfung von Eingaben** hilfreich. Sie kann folgendermaßen gekürzt werden:

```
1 print("Passwort-Checker:\n")
2 passwort="geheim"
3 while True:
4     eingabe=input("Passwort: ")
5     if eingabe!=passwort:
6         print("Eingabe falsch!")
7     else:
8         break
9 print("Das Passwort ist richtig!")
```

```
>>> %Run -c $EDITOR_CONTENT

  Passwort-Checker:

  Passwort: Hallo
  Eingabe falsch!
  Passwort: geheim
  Das Passwort ist richtig!
```

c) Oft sind **verkürzte Darstellungen** möglich um das Skript zu **vereinfachen**:

```
1 print("Passwort-Checker:\n")
2 passwort="geheim"
3 while input("Passwort: ") !=passwort:
4     print("Eingabe falsch!")
5 print("Das Passwort ist richtig!")
```

```
>>> %Run -c $EDITOR_CONTENT

  Passwort-Checker:

  Passwort: Hallo
  Eingabe falsch!
  Passwort: geheim
  Das Passwort ist richtig!
```

Python Programmierung (Grundlagen)

AB-08: Unterprogramme

Funktionen:

a) Um die Übersichtlichkeit zu erhöhen und **sich wiederholende Vorgänge** auszulagern verwendet man Unterprogramme. Man spricht in Python hier auch von **Funktionen**.

```
1  def hallo():
2      print("Hallo Welt!")
3  hallo()
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Hallo Welt!
```

Die Funktion wird mit **def** angekündigt. In den **Klammern** werden die **notwendigen Parameter** angegeben. Man kann einer Funktion **keinen**, **einen** oder **mehrere Parameter** übergeben. Die Funktion wird im Skript immer **definiert bevor** sie **aufgerufen** wird.

b) Hier hat die Funktion den **Parameter** „eingabe“. Der Parameter hat diese Bezeichnung nur innerhalb der Funktion. **Beim Aufruf** wird der Funktion die Variable „name“ als **Argument mitgegeben**, das in der Funktion als Parameter „eingabe“ verarbeitet wird.

```
1  print("Funktionen-Demo\n")
2  def hallo(eingabe):
3      print(f"Hallo {eingabe}!")
4  name=input("Namen eingeben: ")
5  hallo(name)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Funktionen-Demo

  Namen eingeben: Peter
  Hallo Peter!
```

c) Hier werden **zwei Argumente** mitgegeben, in der Funktion addiert und ausgegeben.

```
1  print("Additions-Demo\n")
2  def addieren(a,b):
3      print(a+b)
4  zahl1=int(input("Erste Zahl:"))
5  zahl2=int(input("Zweite Zahl:"))
6  addieren(zahl1,zahl2)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Additions-Demo

  Erste Zahl:4
  Zweite Zahl:5
  9
```

d) Bei einem **Rückgabewert** (return) muss dieser **in einer Variablen gespeichert** werden.

```
1  print("Additions-Demo\n")
2  def addieren(a,b):
3      return a+b
4  zahl1=int(input("Erste Zahl:"))
5  zahl2=int(input("Zweite Zahl:"))
6  ergebnis=addieren(zahl1,zahl2)
7  print(ergebnis)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Additions-Demo

  Erste Zahl:4
  Zweite Zahl:5
  9
```

Python Programmierung (Grundlagen)

AB-09: Eingaben prüfen mit try/except

Eingaben prüfen:

a) Eingaben müssen geprüft werden. Falls eine **Zahl erwartet** wird, aber ein **String** kommt, wird die Ausführung des Skripts mit einem Fehler (**exception**) abgebrochen.

```
1 alter=int(input("Alter eingeben:"))
2 print(f"Du bist {alter} Jahre alt!")
```

```
>>> %Run -c $EDITOR_CONTENT

  Alter eingeben:r
  Traceback (most recent call last):
    File "<string>", line 1, in <module>
  ValueError: invalid literal for int() with base 10: 'r'
```

```
>>> %Run -c $EDITOR_CONTENT

  Alter eingeben:40
  Du bist 40 Jahre alt!
```

b) Für eine erfolgreiche **Prüfung** der Eingabe brauchen wir **try/except**. Hier wird der **Block nach try** versuchsweise ausgeführt und **bei einem Fehler zu except** gesprungen. Eine Korrektur der Eingabe kann nicht vorgenommen werden, weil das Skript endet.

```
1 try:
2     alter=int(input("Alter eingeben:"))
3     print(f"Du bist {alter} Jahre alt!")
4 except:
5     print("Falsche Eingabe")
```

```
>>> %Run -c $EDITOR_CONTENT

  Alter eingeben:f
  Falsche Eingabe
```

c) Mit einer **while-Schleife** wird so lange wiederholt, bis die Eingabe ohne Fehler erfolgt.

```
1 while True:
2     try:
3         alter=int(input("Alter eingeben:"))
4         print(f"Du bist {alter} Jahre alt!")
5         break
6     except:
7         print("Falsche Eingabe")
```

```
>>> %Run -c $EDITOR_CONTENT

  Alter eingeben:f
  Falsche Eingabe
  Alter eingeben:40
  Du bist 40 Jahre alt!
```

d) Optimal hier: Eine **Funktion**, die Eingaben prüft und wiederverwendet werden kann.

```
1 def read_input(eingabe):
2     while True:
3         try:
4             return int(input(eingabe))
5         except:
6             print("Falsche Eingabe")
7 alter=read_input("Alter eingeben:")
8 print(f"Du bist {alter} Jahre alt!")
```

```
>>> %Run -c $EDITOR_CONTENT

  Alter eingeben:z
  Falsche Eingabe
  Alter eingeben:16
  Du bist 16 Jahre alt!
```

Python Programmierung (Grundlagen)

AB-10: Zufallszahlen

Random:

a) Zufallsereignisse machen Programm bewusst unvorhersehbar und variabel. Um Zufallszahlen zu generieren gibt es das Modul **random**. Es muss importiert werden.

```
1 import random
2 for i in range(3):
3     x=random.random()
4     print(round(x,2))
```

→

```
>>> %Run -c $EDITOR_CONTENT

  0.47
  0.02
  0.86
```

Die Funktion generiert **Zahlen im Bereich 0-1**. Ohne Rundung sind es 16 Kommastellen.

b) Randint generiert **ganze Zahlen**, die Argumente sind Start- / Endwert des Bereiches.

```
1 import random
2 for i in range(3):
3     x=random.randint(1,10)
4     print(x)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  10
  9
  7
```

c) Uniform liefert **float**-Werte. Ohne Rundung sind es auch hier 16 Nachkommastellen.

```
1 import random
2 for i in range(3):
3     x=random.uniform(1,10)
4     print(round(x,2))
```

→

```
>>> %Run -c $EDITOR_CONTENT

  1.82
  6.85
  8.88
```

d) Mit **choice** lassen sich zufällige Elemente aus einer Liste / Menge auswählen.

```
1 import random
2 liste=["A","B","C","D","E","F","G","H"]
3 for i in range(3):
4     x=random.choice(liste)
5     print(x)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  E
  G
  F
```

e) sample wählt eine Menge **k** aus einer Liste / einer Zahlenmenge **ohne Zurücklegen**.

```
1 import random
2 liste=["A","B","C","D","E","F","G","H"]
3 x=random.sample(liste,k=4)
4 print(x)
```

```
>>> %Run -c $EDITOR_CONTENT

  ['D', 'C', 'G', 'H']
```

07.2026 (Sch)

Python Programmierung (Grundlagen)

AB-11: Listen

Grundlegende Listenoperationen:

a) Um Elemente mit verschiedenen Datentypen zu speichern braucht man **Listen**.

```
1 liste=["A","B","C","D"]
2 print(liste)
3 print(liste[0],liste[2])
```

```
>>> %Run -c $EDITOR_CONTENT

  ['A', 'B', 'C', 'D']
A C
```

Es können **ganze Listen** oder **Elemente** ausgegeben werden. Erstes Element=**0**!

b) Eine Liste kann problemlos **verschiedene Datentypen** enthalten.

```
1 liste=["A",13,0.5,True]
2 print(liste)
3 print(liste[0],liste[2])
```

```
>>> %Run -c $EDITOR_CONTENT

  ['A', 13, 0.5, True]
A 0.5
```

c) Mit **append** werden Elemente am Ende der Liste **hinzugefügt**.

```
1 liste=["A",13,0.5,True]
2 liste.append("B")
3 print(liste)
```

```
>>> %Run -c $EDITOR_CONTENT

  ['A', 13, 0.5, True, 'B']
```

d) Die Funktion **insert** fügt ein Element an einer **bestimmten Position** (hier: 2) ein.

```
1 liste=["A",13,0.5,True]
2 liste.insert(2,"B")
3 print(liste)
```

```
>>> %Run -c $EDITOR_CONTENT

  ['A', 13, 'B', 0.5, True]
```

e) Mit **pop** wird ein bestimmtes Element gelöscht (hier: 2), mit **clear()** die gesamte Liste.

```
1 liste=["A",13,0.5,True]
2 liste.pop(2)
3 print(liste)
```

```
>>> %Run -c $EDITOR_CONTENT

  ['A', 13, True]
```

f) Mit **sort** kann man eine Liste **alphabetisch sortieren**.

```
1 liste=[]
2 anzahl=3
3 for i in range (1,anzahl+1):
4     eingabe=input(f"Gib Nr.{i}/{anzahl} ein:")
5     liste.append(eingabe)
6 liste.sort()
7 print(liste)
```

```
>>> %Run -c $EDITOR_CONTENT

Gib Nr.1/3 ein:Banane
Gib Nr.2/3 ein:Kiwi
Gib Nr.3/3 ein:Apfel
['Apfel', 'Banane', 'Kiwi']
```

07.2026 (Sch)

Python Programmierung (Grundlagen)

AB-12: Modulo-Rechnung

Ganzzahlig teilen mit Rest:

a) Bei der **Division** ganzer Zahlen können Reste übrigbleiben. Beispiel: 15:2=7 (Rest 1). Diesen **Rest** kann man mit dem **Operator %** (**Modulo**) in Python erfassen: **15%2=1**

```
1  x=15%2
2  print(x)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  1
```

Dabei ist das Ergebnis der Division nicht interessant, sondern **nur der Rest**.

b) Auf diese Weise kann man **gerade** und **ungerade** Zahlen **unterscheiden**.

```
1  def get_int(eingabe):
2      while True:
3          try:
4              return int(input(eingabe))
5          except:
6              print("Falsche Eingabe")
7  zahl = get_int("Gib eine Zahl ein:")
8  if zahl % 2 == 0:
9      print("Die Zahl ist gerade!")
10 else:
11     print("Die Zahl ist ungerade!")
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Gib eine Zahl ein:5
  Die Zahl ist ungerade!
```

c) In einer **Schleife** kann man so beispielweise **jeden fünften Durchgang ausgeben**.

```
1  for i in range (1,21):
2      if i%5==0:
3          print(i)
```

→

```
>>> %Run -c $EDITOR_CONTENT

  5
  10
  15
  20
```

d) Modulo wird hier genutzt um **Elemente regemäßig** zu **wiederholen**.

```
1  farben = ["Rot", "Grün", "Blau"]
2
3  for i in range(6):
4      print(farben[i % 3])
```

→

```
>>> %Run -c $EDITOR_CONTENT

  Rot
  Grün
  Blau
  Rot
  Grün
  Blau
```

07.2026 (Sch)

Python Programmierung (Grundlagen)

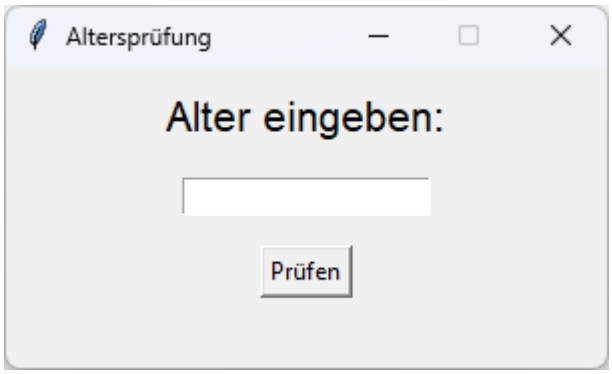
AB-13: GUI-Programmierung

Anwendungen mit Fenster (GUI):

Aus dem **CLI**-Programm in a) kann man mit tkinter leicht ein **GUI**-Programm machen:

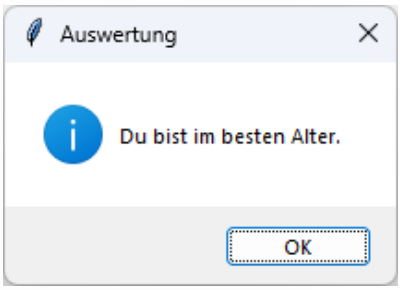
a)

```python
def alter_pruefen(alter):
    if alter < 18:
        return "Du bist noch sehr jung."
    elif alter < 80:
        return "Du bist im besten Alter."
    else:
        return "Du bist schon recht alt."

while True:
    try:
        alter = int(input("Alter eingeben:"))
        print(alter_pruefen(alter))
        break
    except:
        print("Falsche Eingabe")
```



b)

```python
import tkinter as tk
from tkinter import messagebox

def alter_pruefen(alter):
    if alter < 18:
        return "Du bist noch sehr jung."
    elif alter < 80:
        return "Du bist im besten Alter."
    else:
        return "Du bist schon recht alt."

def button_klick():
    try:
        alter = int(edit.get())
        text = alter_pruefen(alter)
        messagebox.showinfo("Auswertung", text)
    except ValueError:
        messagebox.showerror("Fehler", "Falsche Eingabe!")

fenster = tk.Tk()
fenster.title("Altersprüfung")
fenster.geometry("300x150")
fenster.resizable(False, False)
label = tk.Label(fenster, text="Alter eingeben:", font=("Arial", 16))
label.pack(pady=10)
edit = tk.Entry(fenster)
edit.pack(pady=5)
edit.bind("<Return>", lambda event: button_klick())
button = tk.Button(fenster, text="Prüfen", command=button_klick)
button.pack(pady=10)
fenster.mainloop()
```



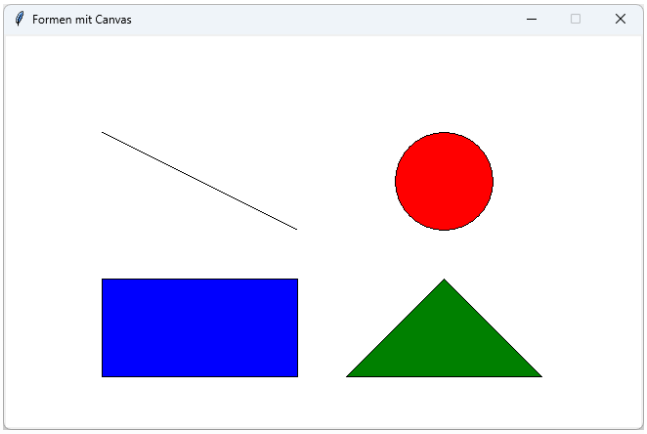
07.2026 (Sch)

Python Programmierung (Grundlagen)

AB-14: Grafik-Darstellung

Tkinter Canvas:

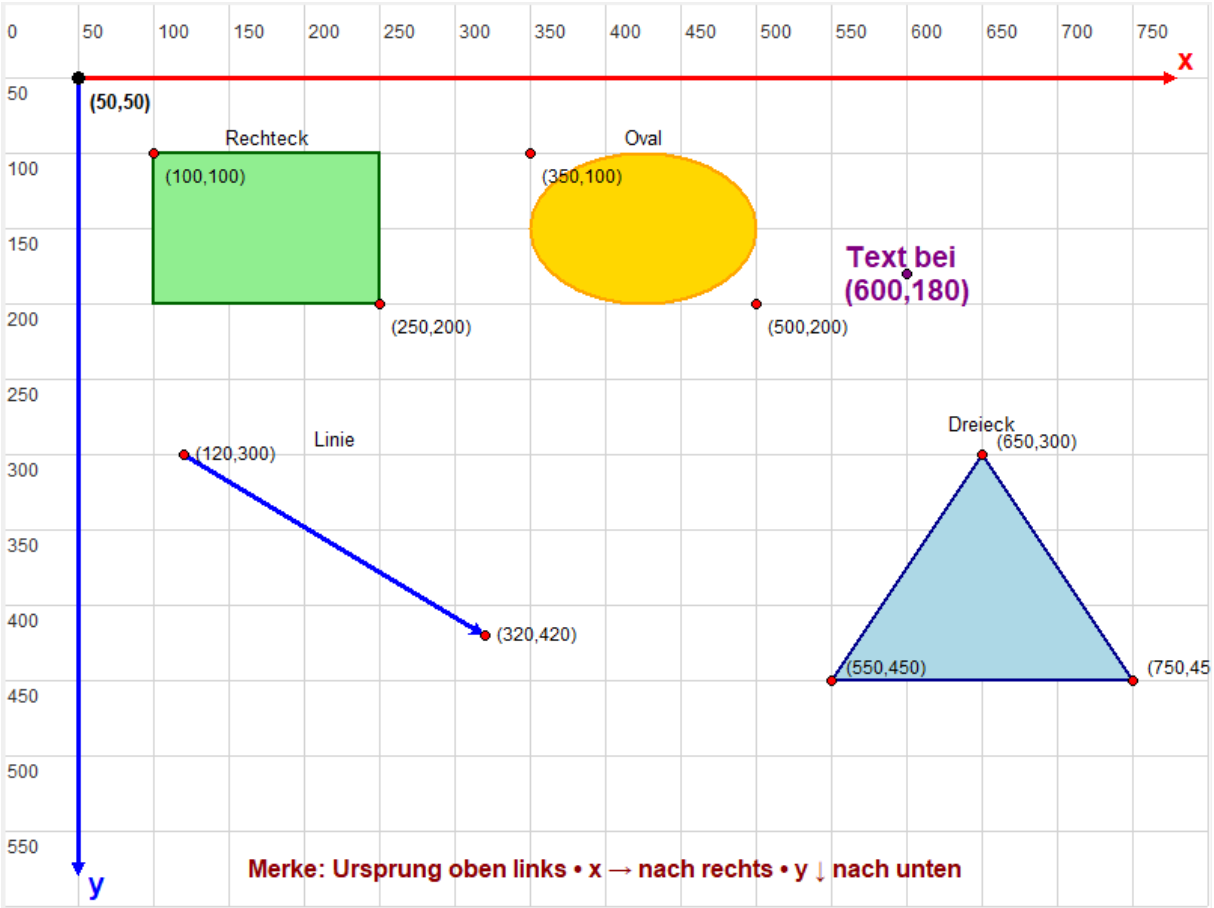
Grundlegende Formen lassen sich relativ einfach **mit Tkinter Canvas** erstellen. Der Code bleibt dabei überschaubar.



a)

```
import tkinter as tk

fenster = tk.Tk()
fenster.title("Formen mit Canvas")
fenster.resizable(False, False)

canvas = tk.Canvas(fenster, width=650, height=400, bg="white")
canvas.pack()

canvas.create_line(100, 100, 300, 200)
canvas.create_oval(400, 100, 500, 200, fill="red", outline="black")
canvas.create_rectangle(100, 250, 300, 350, fill="blue", outline="black")
canvas.create_polygon(350, 350, 450, 250, 550, 350, fill="green", outline="black")

fenster.mainloop()
```

➔ Hilfreich ist hier Wissen über das **Koordinatensystem** in Tkinter.



07.2026 (Sch)

Python Programmierung (Grundlagen)

AB-15: Objekte, Klassen und Vererbung

Objekte und Klassen verstehen:

Eigenschaften und **Funktionen** kann man **in einem Objekt bündeln**. Ein Objekt kann z.B. ein „Held“ sein: Er hat einen Namen, Lebenspunkte und kann kämpfen. In Python ist fast alles ein Objekt. Die Variable „zahl“ mit dem Wert 42 ist ein **Objekt** der **Klasse** „int“.

```
1 zahl=42
2 print(type(zahl))
```

```
>>> %Run -c $EDITOR_CONTENT
  <class 'int'>
```

a) Objekte müssen **auf Basis einer Klasse** (Class) erzeugt werden. Die Klasse ist der Bauplan des Objektes und beschreibt welche Eigenschaften und Funktionen es hat.

```
1 class Held:
2     def __init__(self, name, lebenspunkte):
3         self.name = name
4         self.lebenspunkte = lebenspunkte
5     def kämpfen(self):
6         print(f"Der Held {self.name} ist ein starker Kämpfer!")
```

Klassen werden wie Funktionen (Methoden) zu Beginn des Skriptes abgelegt. Eine Klasse enthält meist einen **Konstruktor** (**__init__**) der automatisch aufgerufen wird, sobald ein neues Objekt erzeugt wird. Der Konstruktor legt die Startwerte eines Objektes fest.

b) Ein **konkretes Objekt** (**Instanz**) existiert aber erst dann, wenn es erzeugt wird: Die Eigenschaften werden in Variablen hinterlegt und dann dem Objekt „held“ übergeben.

```
7 name = "Peter"
8 lebenspunkte = 100
9 held = Held(name, lebenspunkte)
```

Die Variable „held“ speichert eine Referenz auf das erzeugte Objekt. Über diese Referenz kann auf Eigenschaften und Funktionen des Objektes zugegriffen werden.

c) Eigenschaften können anschließend verändert, und Funktionen aufgerufen werden:

```
10 print(f"{held.name} hat {held.lebenspunkte} HP.")
11 held.lebenspunkte=input(f"Neue Lebenspunkte von {held.name}:")
12 print(f"{held.name} hat jetzt {held.lebenspunkte} HP.")
13 held.kämpfen()
```

d) Etwas kompakter, ohne Variablen und mit Standardwerten geht es so:

```
1  class Held:
2      def __init__(self, name="", lebenspunkte=0):
3          self.name = name
4          self.lebenspunkte = lebenspunkte
5      def kämpfen(self):
6          print(f"Der Held {self.name} ist ein starker Kämpfer!")
7  held = Held("Peter", 100)
8  print(f"{held.name} hat {held.lebenspunkte} HP.")
9  held.lebenspunkte=input(f"Neue Lebenspunkte von {held.name}:")
10 print(f"{held.name} hat jetzt {held.lebenspunkte} HP.")
11 held.kämpfen()
```

Python Programmierung (Grundlagen)

AB-15: Objekte, Klassen und Vererbung

Vererbung:

Eine Klasse kann die **Eigenschaften** und **Fähigkeiten** einer anderen Klasse **übernehmen**. Die neue Klasse (Kindklasse) muss nur noch die Unterschiede ergänzen.

a) Die Klasse „Krieger“ erbt die Eigenschaften von „Held“, hat aber mehr Fähigkeiten:

```
1  class Held:
2      def __init__(self, name="", lebenspunkte=0):
3          self.name = name
4          self.lebenspunkte = lebenspunkte
5      def kämpfen(self):
6          print(f"Der Held {self.name} ist ein starker Kämpfer!")
7  class Krieger(Held):
8      def schwertangriff(self):
9          print(f"{self.name} führt einen mächtigen Schwertangriff aus!")
10 krieger = Krieger("Peter", 100)
11 print(f"{krieger.name} hat {krieger.lebenspunkte} HP.")
12 krieger.lebenspunkte=input(f"Neue Lebenspunkte von {krieger.name}:")
13 print(f"{krieger.name} hat jetzt {krieger.lebenspunkte} HP.")
14 krieger.kämpfen()
15 krieger.schwertangriff()
```

```
>>> %Run -c $EDITOR_CONTENT

  Peter hat 100 HP.
  Neue Lebenspunkte von Peter:41
  Peter hat jetzt 41 HP.
  Der Held Peter ist ein starker Kämpfer!
  Peter führt einen mächtigen Schwertangriff aus!
```

b) Eine **Kindklasse** darf **nicht leer** sein. Sie muss mindestens „**pass**“ enthalten.

```
1  class Held:
2      def __init__(self, name="", lebenspunkte=0):
3          self.name = name
4          self.lebenspunkte = lebenspunkte
5      def kämpfen(self):
6          print(f"Der Held {self.name} ist ein starker Kämpfer!")
7  class Krieger(Held):
8      pass
9  krieger = Krieger("Peter", 100)
10 print(f"{krieger.name} hat {krieger.lebenspunkte} HP.")
11 krieger.lebenspunkte=input(f"Neue Lebenspunkte von {krieger.name}:")
12 print(f"{krieger.name} hat jetzt {krieger.lebenspunkte} HP.")
13 krieger.kämpfen()
```

→ Funktionen von Kindklassen können Funktionen von Elternklassen überschreiben!

07.2026 (Sch)