Themen

weitere Themen siehe Seite 73

Themen

Operatoren

Operatoren werden in Programmiersprachen wie Python genutzt, um Variablen Werte zuzuweisen und um in Variablen gespeicherte Werte zu verarbeiten.

Zuweisungsoperatoren

Das Gleichheitszeichen dient als Zuweisungsoperator. Dank des Gleichheitszeichens ist es möglich, einer Variablen einen Wert zuzuweisen.

Berechnungsoperatoren

Die Berechnungsoperatoren werden für die bekannten Grundrechenarten verwendet. Daneben können mit dem Operator ** Zahlen potenziert werden.

Der Operator // liefert als Ergebnis der Division eine ganze Zahl. Auch beim Modulo-Operator % wird eine Division ausgeführt. Als Ergebnis erhält man den Rest der Division.

`+`	Addition	`10 + 3 = 13`
`-`	Subtraktion	`10 - 3 = 7`
`*`	Multiplikation	`10 * 3 = 30`
`**`	Potenzieren	`10 ** 3 = 1000`
`/`	Division	`10 / 3 = 3.33333333333`
`//`	Ganzzahldivision	`10 // 3 = 3`
`%`	Modulo-Operator	`10 % 3 = 1`

Vergleichsoperatoren

Vergleichsoperatoren werden in Abfragen genutzt, um die Bedingung zu formulieren. Das Ergebnis dieser Abfragen lautet entweder `true` oder `false`.

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

Logische Operatoren

Mit Hilfe der logischen Operatoren `and`, `or` und `not` lassen sich mehrere Bedingungen miteinander verknüpfen.

Das Ergebnis ist wahr (`true`), wenn ...	
`and`	... beide Bedingungen erfüllt sind.
`or`	... eine von mehreren Bedingungen erfüllt ist.
`not`	... die betreffende Bedingung nicht erfüllt ist.

Teilmengenoperatoren

Der Teilmengenoperator `in` wird eingesetzt, um zu überprüfen, ob ein Element Teil einer Menge ist. Beispielsweise lässt sich damit feststellen, ob eine Liste ein bestimmtes Wort oder eine bestimmte Zahl enthält. Als Ergebnis erhält man `true` oder `false`.

```
"A" in ["A","B","C"]
"A" not in ["A","B","C"]
```

Rangfolge der Operatoren

Beim Rechnen kennen wir die Regel „Punktrechnung vor Strichrechnung“. Auch bei den Operatoren in Python gibt es eine Reihenfolge, in der sie ausgewertet werden. Die folgende Tabelle zeigt die Operatoren und die Priorität ihrer Abarbeitung.

Priorität		
1	`**`	Potenzieren
2	`*, /, //, %`	Multiplikation, Division, Modulo
3	`+, -`	Addition, Subtraktion
4	`<, <=, >, >=, !=, ==`	Vergleichsoperatoren
5	`in, not in`	Teilmengenoperatoren

Verkettung mit Hilfe von Operatoren

Mit Hilfe der Operatoren `,` oder `+` können Werte kombiniert werden. Man spricht dabei auch vom „Verketten“ der Werte.

Möchte man Zahlen und Text miteinander über den Operator `+` verketten, muss man die Zahl mit Hilfe der Funktion `str` in eine Zeichenkette umwandeln.

```
name = "Anna"
alter = 15
print(name,"ist",alter,"Jahre alt.")
print(name+"ist"+str(alter)+"Jahre alt.")
```

```
Anna ist 15 Jahre alt.
Annaist15Jahre alt.
```

Es fällt auf, dass die Werte mit dem Operator `+` ohne Leerzeichen verkettet werden, während bei Verwendung des Operators `,` automatisch ein Leerzeichen eingefügt wird.

Operatoren

Aufgabe 1
Welche Ausgaben sind nach Ablauf dieser Programmzeilen im Ausgabefenster zu erwarten?

a)
```
print(100 + 25)
```
```
125
```

b)
```
print(2 ** 5)
```
```
32
```

c)
```
print(80 // 3)
```
```
26
```

d)
```
print(25 % 4)
```
```
1
```

e)
```
print(3 * 5 + 8)
```
```
23
```

f)
```
print(3 ** 3 % 5)
```
```
2
```

g)
```
print(60 / 5 - 2)
```
```
10.0
```

h)
```
print(3 * 15 // 4)
```
```
11
```

Aufgabe 2
Welche Ausgaben sind nach Ablauf dieser Programmzeilen im Ausgabefenster zu erwarten?

a)
```
print(5 in [2, 4, 6, 8])
```
```
False
```

b)
```
print("AB" in "RHABARBER")
```
```
True
```

c)
```
print(10 not in [10, 20, 30, 40])
```
```
False
```

d)
```
print("T" in ["H", "O", "R", "B"])
```
```
False
```

e)
```
print("N" not in "STUTTGART")
```
```
True
```

f)
```
print(1 in [-2, -1, 0, 1, 2])
```
```
True
```

g)
```
print(10**2 not in [500, 1000, 1500])
```
```
True
```

h)
```
print(10 % 3 in [3, 6, 9])
```
```
False
```

Aufgabe 3
Welche Ausgaben sind nach Ablauf dieser Programmzeilen im Ausgabefenster zu erwarten?

a)
```
print(7>5 and 14>7)
```
```
True
```

b)
```
print(3<8 or 3>8)
```
```
True
```

c)
```
print(not 10<12)
```
```
False
```

d)
```
print(20>10 and not 10>5)
```
```
False
```

Operatoren

Aufgabe 4

Welche Ausgabe ist nach Ablauf dieses Programms im Ausgabefenster zu erwarten?

```
berg = "Feldberg"
hoehe = 1277
einheit = "Meter"
gebirge = "Schwarzwald"

print("Der",berg,"im",gebirge)
print("ist"+str(hoehe)+einheit+"hoch.")
```

```
Der Feldberg im Schwarzwald
ist1277Meterhoch.
```

Aufgabe 5

Schreibe ein Programm, das den Namen und den Wohnort abfragt und beides in der Form „X wohnt in Y“ wieder ausgibt.

```
name = input("Gib deinen Namen ein")
ort = input("Gib deinen Wohnort ein")

print(name,"wohnt in",ort)

#oder

print(name+" wohnt in "+ort)
```

Beispiel:

```
Linus wohnt in Wangen im Allgäu
```

Aufgabe 6

Schreibe ein Programm, das
- fünf Variablen mit den folgenden Werten anlegt und
- sie mit dem +-Operator verkettet und ausgibt

Texte: Im Jahr
leben
Milliarden Menschen auf der Erde

Zahlen: 2020
7.8

```
text1 = "Im Jahr "
jahr  = 2020
text2 = " leben "
bevoelkerung  = 7.8
text3 = " Milliarden Menschen auf der Erde"

print(text1+str(jahr)+text2+str(bevoelkerung)+text3)
```

```
Im Jahr 2020 leben 7.8 Milliarden Menschen auf der Erde
```

for-Schleifen

Schleifen werden in Programmen genutzt, um Anweisungen mehrmals ausführen zu lassen. Jede Schleife enthält eine Bedingung, die vor jeder Wiederholung geprüft wird, damit die Schleife nicht endlos läuft.

In Python gibt es zwei Arten von Schleifen, die so genannten for-Schleifen und while-Schleifen.

for-Schleifen

Eine for-Schleife wird verwendet, wenn bekannt ist, wie häufig die zu wiederholenden Anweisungen ausgeführt werden sollen.

Die Steuerung von for-Schleifen erfolgt durch Variablen, die als Zähler dienen. for-Schleifen werden deshalb auch als Zählschleifen bezeichnet.

Die Zählvariablen in for-Schleifen werden üblicherweise mit Kleinbuchstaben wie i, j, k etc. benannt.

for-Schleifen können auch verschachtelt werden. Das heißt, dass in eine äußere for-Schleife mit der Zählvariablen i eine oder mehrere for-Schleifen mit den Zählvariablen j, k usw. eingebettet werden können.

```
for i in (1, 2):
    print(i)
    for j in ("A", "B"):
        print(j)
```

Nach dem einleitenden `for` folgt die Bedingung, bei deren Erfüllung die enthaltenen Anweisungen ausgeführt werden. Diese Zeile wird mit einem Doppelpunkt abgeschlossen, die nachfolgenden Anweisungen werden eingerückt.

In unserem Beispiel kann die Zählvariable i nacheinander die Werte 1 und 2 annehmen. In jedem Durchlauf der äußeren Schleife wird die zweite Schleife durchlaufen. Deren Zählvariable j kann die Werte A und B annehmen. Entsprechend erhält man für unser Beispiel die folgende Ausgabe:

```
1
A
B
2
A
B
```

range()-Funktion

In einfachen Fällen wie in unserem Beispiel lassen sich alle möglichen Werte, die die Zählvariablen annehmen können, direkt in der Bedingung notieren. Ist die Anzahl der möglichen Werte größer, benötigt man eine Funktion, die eine Liste dieser Werte erzeugt.

Hierfür kann die Funktion `range()` in Kombination mit der `list()`-Funktion eingesetzt werden. Sie kann mit einem, zwei oder drei Parametern versehen werden, abhängig davon, welche Werte die Liste enthalten soll. Als Parameter sind ganze Zahlen zulässig, die aber sowohl positiv als auch negativ sein dürfen.

Mit einem Parameter erzeugt man eine Liste ganzer Zahlen, die stets mit Null beginnt und in Einerschritten bis zur Zahl unterhalb des Parameterwertes reicht. Der Parameter selbst ist nicht Teil der Liste.

```
print(list(range(4)))
print(list(range(2,6)))
print(list(range(4,16,4)))
print(list(range(2,-6,-2)))
```

```
[0, 1, 2, 3]
[2, 3, 4, 5]
[4, 8, 12]
[2, 0, -2, -4]
```

Für eine Liste, die nicht mit Null beginnt, muss man die Funktion `range()` mit zwei Parametern verwenden. Das Ergebnis beginnt in diesem Fall mit dem ersten Parameter und endet eins unter dem zweiten Parameter.

Der dritte Parameter in der Funktion `range()` ermöglicht Listen, die nicht in Einerschritten ansteigen. Der erste der drei Parameter ist zugleich der Startwert der Liste. Der dritte Parameter definiert die Schrittweite. Der zweite Parameter markiert den Wert, der gerade nicht mehr Teil der Liste ist. Durch die Verwendung des dritten Parameters sind so auch absteigende Listen möglich.

In for-Schleifen lässt sich mit Hilfe der `range()`-Funktion ein Zahlenbereich erzeugen, den die Zählvariable i durchläuft. In unserem Beispiel wären das 10 Wiederholungen von i = 0 bis i = 9.

```
for i in range(10):
```

for-Schleifen

Aufgabe 1
Welche Ausgaben sind nach Ablauf dieser Programmzeilen im Ausgabefenster zu erwarten?

a)
```
print(list(range(1,10,2)))
```

b)
```
print(list(range(-2,-10,-4)))
```

c)
```
print(list(range(3)))
```

d)
```
print(list(range(15,20)))
```

e)
```
print(list(range(-5,0)))
```

f)
```
print(list(range(12,4,-2)))
```

g)
```
print(list(range(10,90,15)))
```

a)
```
[1, 3, 5, 7, 9]
```

b)
```
[-2, -6]
```

c)
```
[0, 1, 2]
```

d)
```
[15, 16, 17, 18, 19]
```

e)
```
[-5, -4, -3, -2, -1]
```

f)
```
[12, 10, 8, 6]
```

g)
```
[10, 25, 40, 55, 70, 85]
```

Aufgabe 2
Schreibe ein Programm, mit dessen Hilfe die Lottozahlen 6 aus 49 ermittelt und ausgegeben werden können.

Beispiellösung:

```
from random import *
seed()

print(sample(range(1,50),6))
```

Aufgabe 3
Lass den Computer im übertragenen Sinne 6000 mal würfeln und anschließend ausgeben, wie häufig die einzelnen Augenzahlen gefallen sind.

- Weise dazu sechs Variablen den Wert 0 zu.
- Verwende eine for-Schleife, um die Anzahl der gewürfelten Zahlen zu zählen.
- Erhöhe nach jedem Würfeln den Wert der Variable, die zur jeweiligen Augenzahl gehört.
- Gib nach dem Durchlaufen der Schleife aus, wie häufig die einzelnen Ziffern gefallen sind

Beispiellösung:

```
from random import *
seed()

anzahl1 = 0
anzahl2 = 0
anzahl3 = 0
anzahl4 = 0
anzahl5 = 0
anzahl6 = 0

for i in range(6000):
    augenzahl = randint(1,6)

    if augenzahl == 1:
        anzahl1 = anzahl1 + 1
    elif augenzahl == 2:
        anzahl2 = anzahl2 + 1
    elif augenzahl == 3:
        anzahl3 = anzahl3 + 1
    elif augenzahl == 4:
        anzahl4 = anzahl4 + 1
    elif augenzahl == 5:
        anzahl5 = anzahl5 + 1
    else:
        anzahl6 = anzahl6 + 1

print("Eins:",anzahl1)
print("Zwei:",anzahl2)
print("Drei:",anzahl3)
print("Vier:",anzahl4)
print("Fuenf:",anzahl5)
print("Sechs:",anzahl6)
```

for-Schleifen

Aufgabe 4

Schreibe Programme, die die Zeilen- und Spaltenzahl abfragen und anschließend – mit zwei for-Schleifen – die folgenden Muster ausgeben.

In jeder Runde soll dabei nur ein Zeichen angefügt werden.

Die Programme sollen beliebige Zeilen- und Spaltenanzahlen erlauben und entsprechend größere oder kleinere Figuren erzeugen.

Beispiellösungen:

a)
```
OOOOO
OOOOO
OOOOO
OOOOO
OOOOO
```

```
groesse = input("Anzahl Zeilen/Spalten")
ausgabe = ""
for zeile in range(groesse):
    for spalte in range(groesse):
        ausgabe = ausgabe + "O"
    print(ausgabe)
    ausgabe = ""
```

b)
```
O----
-O---
--O--
---O-
----O
```

```
groesse = input("Anzahl Zeilen/Spalten")
ausgabe = ""
for zeile in range(groesse):
    for spalte in range(groesse):
        if spalte == zeile:
            ausgabe = ausgabe + "O"
        else:
            ausgabe = ausgabe + "-"
    print(ausgabe)
    ausgabe = ""
```

c)
```
OOOOO
-OOOO
--OOO
---OO
----O
```

```
groesse = input("Anzahl Zeilen/Spalten")
ausgabe = ""
for zeile in range(groesse):
    for spalte in range(groesse):
        if spalte >= zeile:
            ausgabe = ausgabe + "O"
        else:
            ausgabe = ausgabe + "-"
    print(ausgabe)
    ausgabe = ""
```

d)
```
OOOOO
O---O
O---O
O---O
OOOOO
```

```
zeilen = input("Anzahl Zeilen eingeben")
spalten = input("Anzahl Spalten eingeben")
ausgabe = ""
for zeile in range(zeilen):
    for spalte in range(spalten):
        if zeile == 0 or spalte == 0 or zeile == zeilen-1 or spalte == spalten-1:
            ausgabe = ausgabe + "O"
        else:
            ausgabe = ausgabe + "-"
    print(ausgabe)
    ausgabe = ""
```

Selbst definierte Funktionen

Häufig müssen in Programmen Abfolgen von Anweisungen mehr als einmal ausgeführt werden. Um diese Programmabschnitte nicht wiederholen zu müssen, verwe ndet man Funktionen.

Funktionen werden im oberen Teil eines Programms definiert. Im Hauptprogramm muss dann nur noch der Funktionsname aufgerufen werden, damit die Anweisungen ausgeführt werden.

Längere Programme erhalten dadurch eine übersichtliche Struktur. Sprechende Funktionsnamen können zudem helfen, die Wirkungsweise eines Programms zu verstehen.

Definition mit def

Die Definition einer Funktion beginnt mit der Anweisung **def** und dem Namen der Funktion. Der Name einer Funktion kann frei gewählt werden. Nach dem Namen folgen runde Klammern und ein Doppelpunkt. Die Anweisungen, die zur Funktion gehören, werden eingerückt.

In der Klammer können ein oder mehrere Parameter aufgelistet sein, die innerhalb der Funktion benötigt werden.

```
#Definition der Funktion quadrat
def quadrat(seite):
    ergebnis = seite * seite
    return ergebnis

#Hauptprogramm
seite = input("Seitenlänge")

flaeche = quadrat(seite)
print("Flaeche:",flaeche)
```

Im Hauptprogramm wird die Funktion aufgerufen. In unserem Beispiel wird das Ergebnis der Funktion der Variablen **flaeche** zugewiesen und ausgegeben.

return

Damit das innerhalb der Funktion berechnete Ergebnis im Hauptprogramm zugewiesen und ausgedruckt werden kann, wird die Anweisung **return** benötigt.

return erzeugt einen Rückgabewert, der als Ergebnis der Funktion an das Hauptprogramm zurückgeliefert wird. Die Variable **ergebnis** wird dabei nicht mit übergeben.

Zugleich beendet die Anweisung **return** die Ausführung der Funktion.

Lokale und globale Variablen

Jede Funktion bildet für die Variablen einen geschlossenen, lokalen Namensraum. Das bedeutet, dass man auf Variablen, die innerhalb einer Funktion definiert wurden, aus anderen Funktionen oder dem Hauptprogramm heraus nicht zugreifen kann.

In unserem ersten Beispiel trifft das auf die Variable **ergebnis** zu. Sie wird in der Funktion quadrat definiert. Im Hauptprogramm wird aber eine neue Variable benötigt, um die berechnete Fläche auszugeben.

Möchte man eine Variable sowohl innerhalb einer Funktion als auch im restlichen Programm verwenden, muss sie in der Funktion den Status **global** erhalten.

Mehrere Variablen, die als **global** deklariert und mit **return** übergeben werden sollen, werden – durch Komma getrennt – aneinandergereiht.

```
#Definition der Funktion rechteck
def rechteck(seite_a,seite_b):
    #Status global zuweisen
    global flaeche, umfang
    flaeche = seite_a * seite_b
    umfang = 2 * seite_a + 2 * seite_b
    return flaeche, umfang

#Hauptprogramm
seite_a = input("Seitenlaenge a")
seite_b = input("Seitenlaenge b")

rechteck(seite_a,seite_b)
#Variable aus Funktion
print("Flaeche:",flaeche)
print("Umfang:",umfang)
```

Globale Variablen können nach Ablauf der Funktion auch im Hauptprogramm verwendet und beispielsweise ausgegeben werden.

Selbst definierte Funktionen

Aufgabe 1
Nimm in diesem Programm die folgenden kleinen Veränderungen vor. Beschreibe, was daraufhin beim Ausführen des Programms geschieht, und erkläre, warum das so ist.

```
def quadrat(seite):
    ergebnis = seite * seite
    return ergebnis

seite = input("Seitenlaenge")

flaeche = quadrat(seite)
print(flaeche)
```

a) Kommentiere die Zeile `return ergebnis` aus.

```
def quadrat(seite):
    ergebnis = seite * seite
#   return ergebnis

seite = input("Seitenlaenge")

flaeche = quadrat(seite)
print(flaeche)
```

```
None
```

Anstelle der berechneten Fläche wird „None" ausgegeben.
Wenn die Anweisung return auskommentiert ist, wird das Ergebnis der Flächenberechnung aus der Funktion nicht übergeben. Die Funktion wird dadurch nach ihrem Aufruf im Hauptprogramm zwar ausgeführt, gibt aber keinen Wert zurück. Deshalb erscheint als Ausgabe „None", was soviel bedeutet wie „nichts" oder „kein Wert".

b) Ändere die letzten beiden Zeilen des Hauptprogramms wie folgt:

```
quadrat(seite)
print ergebnis
```

```
def quadrat(seite):
    ergebnis = seite * seite
    return ergebnis

seite = input("Seitenlaenge")

quadrat(seite)
print(ergebnis)
```

```
Der Name 'ergebnis' ist nicht
definiert oder falsch geschrieben.
Es existiert also keine Variable
oder Funktion mit diesem Namen.
```

Beim Ausführen des Programms erscheint die Fehlermeldung, dass keine Variable mit dem Namen „ergebnis" existiert.
Die Variable „ergebnis" wird innerhalb der Funktion definiert und gilt nur lokal innerhalb der Funktion. Es ist daher nicht möglich, im Hauptprogramm auf diese Variable zuzugreifen und sie auszugeben.

c) Fasse die letzten beiden Zeilen des Hauptprogramms zusammen zu

```
print quadrat(seite)
```

```
def quadrat(seite):
    ergebnis = seite * seite
    return ergebnis

seite = input("Seitenlaenge")

print(quadrat(seite))
```

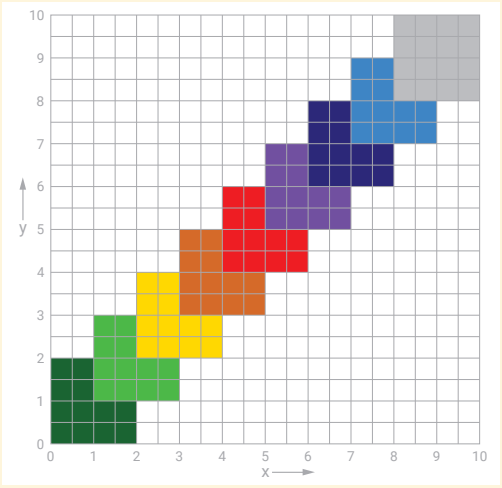
```
25
```

Nach dem Ausführen des Programms und der Eingabe der Seitenlänge 5 erscheint die Ausgabe 25. Durch die Anweisung return wird das Ergebnis der Flächenberechnung als Rückgabewert an das Hauptprogramm übergeben. Dadurch ist es möglich, das Ergebnis im Hauptprogramm auszugeben. Das Ergebnis vor dem Ausgeben einer Variablen „flaeche" zuzuweisen, ist nicht notwendig.

Selbst definierte Funktionen

Aufgabe 2

Erstelle ein Programm, das neun in einer zufälligen RGB-Farbe gefärbte Quadrate auf diese Weise diagonal anordnet.
(Das Kästchenraster dient nur der Orientierung.)



a) Importiere dazu die benötigten Bibliotheken, setze den Anfangswert für den Zufallsgenerator und lass ein Grafikpanel mit den Parametern wie im Bild zeichnen.

b) Definiere eine Funktion, die ein buntes Quadrat zeichnet. Die RGB-Farbe soll aus zufälligen Werten für Rot, Grün und Blau zusammengesetzt werden.

c) Im Hauptprogramm soll zunächst der untere linke Eckpunkt des ersten Quadrats auf 0, 0 gesetzt werden.

d) In einer Schleife sollen dann nacheinander die neun Quadrate gezeichnet werden.
Die oberen rechten Eckpunkte der Quadrate sollen jeweils ausgehend vom aktuellen linken unteren Eckpunkt definiert werden.
Nach dem Ausführen der Funktion sollen die Koordinaten des linken unteren Eckpunktes um 1 nach oben und nach rechts verändert werden.

Beispiellösung:

```
from gpanel import *
from random import *

seed()
makeGPanel(0, 10, 0, 10)

def QuadratZeichnen(x1,y1,x2,y2):
    r = randint(0,255)
    g = randint(0,255)
    b = randint(0,255)
    setColor(r,g,b)
    fillRectangle(x1,y1,x2,y2)

#Hauptprogramm
x1 = 0
y1 = 0

for i in range(1,10):
    QuadratZeichnen(x1,y1,x1+2,y1+2)
    x1 = x1 + 1
    y1 = y1 + 1
```

Die Funktion QuadratZeichnen erwartet vier Parameter, unabhängig davon, ob es Zahlen oder Variablen sind. Die Parameter 3 und 4 werden daher den lokalen Variablen x2 und y2 zugeordnet, obwohl sie anders heißen.

Selbst definierte Funktionen

Aufgabe 3
Erstelle ein Programm, das eingegebene Wörter auf darin enthaltene, zufällig ausgewählte Buchstaben prüft.

a) Importiere dazu die benötigte Bibliothek und setze den Anfangswert für den Zufallsgenerator.

b) Definiere eine Funktion, die einen zufälligen Buchstaben von a bis z auswählt und als Rückgabewert an das Hauptprogramm übergibt.

c) Im Hauptprogramm soll mit Hilfe der Funktion drei Variablen nacheinander jeweils ein ausgewählter Buchstabe zugeordnet werden.

d) Der Nutzer soll zur Eingabe eines Wortes aufgefordert werden. In der Aufforderung sollen die drei Buchstaben enthalten sein, beispielsweise „Wort mit abc eingeben“.

e) Es soll geprüft werden, ob das eingegebene Wort alle drei Buchstaben enthält. Falls das so ist und auch falls das nicht so ist, sollen entsprechende Meldungen ausgegeben werden.

Beispiellösung:

```
from random import *
seed()

#einen Zufallsbuchstaben auswählen
def BuchstabeWaehlen():
    buchstabe = chr(randint(97, 123))
    return buchstabe

#Zufallsbuchstaben den Variablen zuweisen
b1 = BuchstabeWaehlen()
b2 = BuchstabeWaehlen()
b3 = BuchstabeWaehlen()

#Wort eingeben lassen und Buchstaben prüfen
wort = input("Wort mit "+b1+b2+b3+" eingeben")



if b1 and b2 and b3 in wort:  ❶
    print("Super!")
else:
    print("Hast du genau genug hingeschaut?")
```

❶ auch diese Programmzeile ist an dieser Stelle möglich und richtig:

```
if b1 in wort and b2 in wort and b3 in wort:
```

Programmierfehler (Bugs)

Beim Programmieren passieren sehr leicht Fehler. Selbst Profis gelingt es selten, auf Anhieb einen vollkommen fehlerfreien Programmcode zu schreiben. Programmierfehler – Profis nennen sie Bug (von englisch bug „Wanze, Ungeziefer“) – führen zu Fehlfunktionen, Programmabstürzen oder Sicherheitslücken. Das Suchen und Beseitigen von Fehlern ist daher ein wichtiger Schritt in der Softwareentwicklung.

Bei der Ausführung eines Programms innerhalb der Entwicklungsumgebung, wird es durch den so genannten Compiler (von englisch compile „zusammentragen“) in eine Form übersetzt, die vom Computer verstanden und ausgeführt werden kann. Viele häufig vorkommende Fehler werden dabei vom Compiler bereits erkannt und führen zu Fehlermeldungen der Entwicklungsumgebung. Diese Fehler werden daher auch als Compilerfehler bezeichnet. Damit man in den eigenen Programmen Compilerfehler besser findet, hilft es, die wichtigsten Arten von Programmierfehlern zu kennen.

Lexikalische Fehler

Werden in einem Programmcode Zeichen verwendet, die nicht zum verwendeten Alphabet gehören, spricht man von einem lexikalischen Fehler. In unserem Beispiel wird das griechische π als Zeichen erkannt, das nicht zum erlaubten Zeichensatz gehört.

```
radius = 10
flaeche = π * radius ** 2
print(flaeche)
```

```
Ungültiges Zeichen: 'π'/'[960]' [line 2]
```

Syntaktische Fehler (Syntaxfehler)

Jede Programmiersprache hat ihre eigenen Regeln für die richtige Schreibweise der Anweisungen – die Syntax. Sie ist mit den Regeln für Rechtschreibung und Grammatik vergleichbar. Fehlende Doppelpunkte, Einrückungen, Anführungszeichen oder Klammern sind die häufigsten syntaktischen Fehler. In unserem Beispiel ist es ein Operator, der für Zuweisungen nicht zulässig ist.

```
radius = 10
flaeche == 3.14 * radius ** 2
print(flaeche)
```

```
Verwende ein einzelnes Gleichheits-
zeichen '=' für Zuweisungen. [line 2]
```

Semantische Fehler

Die Semantik wird auch als Bedeutungslehre bezeichnet. In einem semantisch korrekten Programmcode sind alle Anweisungen so formuliert, wie es ihrer Bedeutung und der Programmlogik entspricht. Häufige semantische Fehler sind fehlende Variablendefinitionen, eine falsche Reihenfolge der Parameter, das Aufrufen einer Funktion vor deren Definition oder ein fehlendes `return` bei einer Funktion. In unserem Beispiel ist die Variable `pi` nicht definiert.

```
radius = 10
flaeche = pi * radius ** 2
print(flaeche)
```

```
Der Name 'pi' ist nicht definiert
oder falsch geschrieben.
```

Logische Fehler

Programme mit logischen Fehlern laufen ohne Fehlermeldungen durch, liefern aber falsche Ergebnisse. Entsprechend sind diese Fehler am schwersten zu finden. Häufige logische Fehler sind vertauschte Rechenoperationen oder fehlerhafte Berechnungsformeln wie in unserem Beispiel. Auch Denkfehler beim Aufbau von Programmen zählen zu den logischen Fehlern.

```
radius = 10
flaeche = 3.14 * radius * 2
print(flaeche)
```

```
62.8
```

Die Grenzen zwischen den Fehlerarten sind häufig etwas verschwommen. So ist das Komma in der Zahl `3.14` sowohl ein Syntaxfehler als auch ein semantischer Fehler, da er zu einer Fehlinterpretation der Zahl und damit zu einem völlig falschen Ergebnis führt.

```
radius = 10
flaeche = 3,14 * radius ** 2
print(flaeche)
```

```
(3, 1400)
```

Programmierfehler (Bugs)

Warum heißen Programmierfehler „Bug“?

Überall, wo etwas programmiert wird, können natürlich auch Fehler auftreten. Diese Fehler in Computerprogrammen werden häufig als „Bug“ bezeichnet. Das Wort Bug kommt aus dem Englischen und bedeutet Wanze oder Ungeziefer.

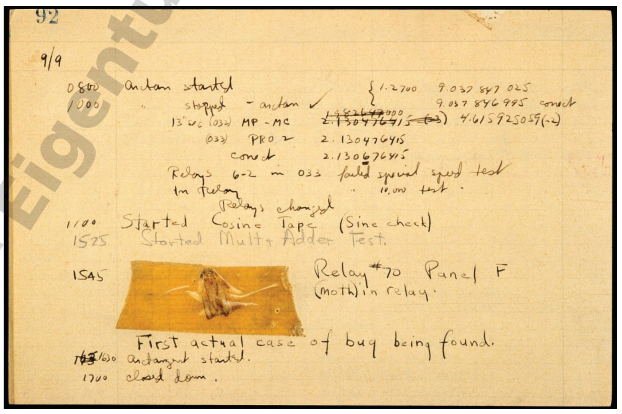
Bereits im 19. Jahrhundert hatte sich offenbar unter amerikanischen Ingenieuren das Wort „Bug“ als Synonym für eine Fehlfunktion oder einen Konstruktionsfehler eingebürgert. Vom Erfinder der Glühlampe, Thomas Alva Edison (1847–1931), ist ein Brief aus dem Jahre 1878 überliefert, in dem er das Wort „bug“ im Zusammenhang mit technischen Problemen und Fehlern verwendet. Dem Gebrauch des Wortes liegt die (scherzhafte) Vorstellung zugrunde, dass sich ein kleines Krabbeltier an den Zahnrädern des Getriebes, an den Leitungen etc. zu schaffen macht.

Als nach dem Zweiten Weltkrieg das Zeitalter der Computer begann, übernahmen die Programmierer und Computerexperten das Wort aus dem Ingenieurjargon. Speziell die folgende Episode ist in die Geschichte eingegangen: An der Harvard University wurde 1947 der Mark-II-Rechner fertiggestellt, ein elektromechanischer Computer auf Basis elektromagnetischer Relais. Er wog stolze 23 Tonnen und benötigte 0,75 Sekunden für eine einfache Multiplikation. Der leistungsfähigste Computer seiner Zeit sollte komplizierte Rechnungen für das US-Militär ausführen.

Doch am 9. September 1947 unterliefen der sündhaft teuren Maschine plötzlich peinliche Rechenfehler. Grace Hopper, eine Computerwissenschaftlerin der ersten Stunde, und ihre Techniker suchten nach dem Fehler, doch das Programm war einwandfrei. Schließlich fanden sie nach fünf Stunden eine verendete Motte, die sich im Relais mit der Nummer 70 verklemmt hatte.

Dadurch konnte das Relais nicht richtig schließen und verursachte die Rechenfehler. Nachdem die Motte entfernt war, funktionierte der Computer wieder tadellos.

Grace Hopper und ihre Techniker dokumentierten den Fehler und seine Ursache im Logbuch des Computerlabors: „First actual case of bug being found“ („Das erste Mal, dass tatsächlich ein Ungeziefer gefunden wurde“). Die Motte klebten sie mit einem Klebestreifen daneben. Die Logbuchseite wird bis heute im Smithsonian Institute, dem Nationalmuseum der USA, aufbewahrt.



Logbuch von Grace Hopper im National Museum of American History (https://americanhistory.si.edu/collections/search/object/nmah_334663, Stand März 2021)

Dr. Grace Hopper wurde erst im Alter von 80 Jahren im Dienstgrad eines Flottillenadmirals in den Ruhestand entlassen. Sie trug auch den Spitznamen „Grandma COBOL“ („Großmutter COBOL“), da sie wesentliche Vorarbeiten zur Entwicklung der Programmiersprache COBOL geleistet hatte. Ihr verdanken wir, dass wir heute Computerprogramme in einer verständlichen Sprache verfassen können, statt nur mit Einsen und Nullen.

Programmierfehler (Bugs)

Aufgabe 1

a) Markiere alle Fehler in diesem Programm.

b) Notiere jeweils die Fehlermeldung und die Art der Fehler.

```
from random import
from GPanel import *
seed(
makegpanel(0, 10, 0, 10)
def KreisZeichnen(r)
    rot = randint(0,255)
    grün = randint(0,255)
    blau = randint(0,255)
    setColor(rot,gruen,blau)
    fillCircle(R)

Hauptprogramm
x = 0
y == 0
for i in range1,5):
x = x + i
    y = y + i
    r = i
    pos x,y)
    KreisZeichnen(r)
```

Richtiges Programm:

```
from random import *
from gpanel import *
seed()

makeGPanel(0, 10, 0, 10)

def KreisZeichnen(r):
    rot = randint(0,255)
    gruen = randint(0,255)
    blau = randint(0,255)
    setColor(rot,gruen,blau)
    fillCircle(r)

#Hauptprogramm
x = 0
y = 0

for i in range(1,5):
    x = x + i
    y = y + i
    r = i
    pos(x,y)
    KreisZeichnen(r)
```

Fehlerpositionen, Fehlerarten und Fehlermeldungen:

```
from random import )  Syntaxfehler
Unvollständige 'import'-Anweisung.
from GPanel import *  Semantischer Fehler
Python findet kein Modul mit dem Namen GPanel.
seed(  Syntaxfehler
Fehlende schliessende Klammer: ')'

makegpanel(0, 10, 0, 10)  Semantischer Fehler
Der Name 'makegpanel' ist nicht definiert
oder falsch geschrieben.

def KreisZeichnen(r)  Syntaxfehler
Doppelpunkt ':' fehlt.
    rot = randint(0,255)
    grün = randint(0,255)  Lexikalischer Fehler
Ungültiges Zeichen: 'ü'/'
    blau = randint(0,255)
    setColor(rot,gruen,blau)
    fillCircle(R)  Semantischer Fehler
Der Name 'R' ist nicht definiert oder
falsch geschrieben.

Hauptprogramm  Syntaxfehler
Wirkungslose Anweisung.
x = 0
y == 0  Syntaxfehler
Verwende ein einzelnes Gleichheits-
zeichen '=' für Zuweisungen.

for i in range1,5):  Syntaxfehler
Unerwartete(s) Symbol(e): ')'
x = x + i  Syntaxfehler
Hier fehlt der Code-Block oder er ist
nicht korrekt eingerückt.
    y = y + i
    r = i
    pos x,y)  Syntaxfehler
Da scheinen Klammern zu fehlen.
    KreisZeichnen(r)
```

Programmierfehler (Bugs)

Aufgabe 2

Im folgenden Programm sind sechs Zeilen markiert, deren Fehlen im weiteren Verlauf des Programms einen semantischen Fehler verursacht.

Beschreibe für jede dieser markierten Zeilen, was passiert, wenn sie fehlt und wo sich dieser Fehler auswirkt.

```
1   from gpanel import *  (1)
2   makeGPanel(-10, 10, -6, 14)  (2)
3   def BogenZeichnen(radius):  (3)
4       fillArc(radius,0,180)
5   farben = ["Red","DarkOrange","Gold",
    "LimeGreen","RoyalBlue","BlueViolet",
    "DeepPink","WhiteSmoke"]  (4)
6   for i in range(0,8):
7       farbe = farben[i]  (5)
8       setColor(farbe)
9       radius = 9 - i  (6)
10      BogenZeichnen(radius)
```

(1) Wenn das Modul gpanel nicht importiert wird, stehen die in diesem Modul definierten Funktionen nicht zur Verfügung.
Damit ist die Ausführung folgender Anweisungen nicht möglich:
– das Grafikpanel (Zeile 2) kann nicht erstellt werden,
– der Bogen (Zeile 4) kann nicht gezeichnet werden
– die Farbe (Zeile 8) kann nicht gesetzt werden

(2) Wenn das Grafkpanel nicht erstellt wird, kann der Bogen (Zeile 4) nicht gezeichnet werden.

(3) Wenn die Funktion „BogenZeichnen(radius)" nicht definiert wird, kann sie in Zeile 10 nicht aufgerufen werden.

(4) Wenn die Liste der Farben nicht definiert wird, kann in Zeile 7 nicht eine Farbe aus der Liste aufgerufen werden.

(5) Wenn die Variable „farbe" nicht definiert wird, kann sie im Befehl setColor (Zeile 8) nicht aufgerufen werden.

(6) Wenn die Variable „radius" nicht definiert wird, kann sie im Funktionsaufruf (Zeile 10) nicht verwendet werden.

Programmierfehler (Bugs)

Aufgabe 3

Das folgende Programm soll dieses Gesicht zeichnen.

a) Markiere die logischen Fehler im Programm, die das gewünschte Ergebnis verhindern.

b) Korrigiere das Programm.

c) Übertrage deine Korrekturen in das Programm Aufgabe_13-3_Gesicht.py. Liefert das Programm nun das gewünschte Ergebnis?

```
from gpanel import *
makeGPanel(-10, 10, -10, 10)

rot = 10
gruen = 150
blau = 200
setColor(blau, gruen, rot)
fillCircle(5)

pos(-2.5, 2)
pos(2.5, 2)
fillCircle(1)
fillCircle(1)
setColor("white")
fillTriangle(-0.75,-0.5,0.75,-0.5,0,1)
pos(-1.5, 0)
fillArc(2.5, 0, 180)
```

falsche Reihenfolge

falsche Reihenfolge x, y

falscher Startwinkel, falsche Bogenrichtung

Richtiges Programm:

```
from gpanel import *
makeGPanel(-10, 10, -10, 10)

rot = 10
gruen = 150
blau = 200
setColor(rot, gruen, blau)
fillCircle(5)

setColor("white")
pos(-2.5, 2)
fillCircle(1)
pos(2.5, 2)
fillCircle(1)
fillTriangle(-0.75,-0.5,0.75,-0.5,0,1)
pos(0, -1.5)
fillArc(2.5, -0, -180)
```

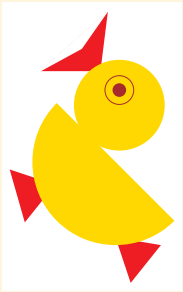
Programmierfehler (Bugs)

Aufgabe 4

Das Programm Aufgabe_13-4_Kueken.py enthält in fast jeder Zeile einen Syntaxfehler.

Finde und korrigiere die Fehler.

Führe das Programm testweise aus. Zeichnet es das Küken?



```
from gpanel import
makeGpanel(-10, 10, -10, 10)

#Fuesse
setColor("red)
fillTriangle(0,-5, 1,-8, 3,-5.5)
filltriangle(-7,-0.5, -6,-4, -4;-1.5)

Schnabel
filTriangle(-5,6, -0.5,10, -1,6)
setColor("white"
fillTriangle(-5,6, -0.5,10, -2.25,7.5))

#Koerper und Kopf
setColour("gold")
fill Arc(5.5, -45, -180)
pos (0.25, 3.75)
fillCircel(3)

#Auge
setColor"brown")
pos(0.25, 4.75)
lineWide(3)
Circle(1)
fillCircle(0.5
```

Richtiges Programm:

```
from gpanel import *
makeGPanel(-10, 10, -10, 10)

#Fuesse
setColor("red")
fillTriangle(0,-5, 1,-8, 3,-5.5)
fillTriangle(-7,-0.5, -6,-4, -4,-1.5)

#Schnabel
fillTriangle(-5,6, -0.5,10, -1,6)
setColor("white")
fillTriangle(-5,6, -0.5,10, -2.25,7.5)

#Koerper und Kopf
setColor("gold")
fillArc(5.5, -45, -180)
pos(0.25, 3.75)
fillCircle(3)

#Auge
setColor("brown")
pos(0.25, 4.75)
lineWidth(3)
circle(1)
fillCircle(0.5)
```