UNTERPROGRAMMTECHNIK

Divide et impera

WIEDERHOLUNG

- Zeichnen eines Quadrats

```
import turtle
laenge=int(input("Seitenlänge?"))

alex=turtle.Turtle()
alex.fd(laenge)
alex.rt(90)
alex.fd(laenge)
alex.rt(90)
alex.fd(laenge)
alex.rt(90)
alex.fd(laenge)
alex.rt(90)
turtle.exitonclick()
```

```
alex=turtle.Turtle()
for _ in range(0,4):
    alex.fd(laenge)
    alex.rt(90)
turtle.exitonclick()
```

PROBLEM

- Wir brauchen nicht 1 Quadrat, sondern 10…

```
import turtle
laenge=int(input("Seitenlänge?"))

alex=turtle.Turtle()
alex.fd(laenge)
alex.rt(90)
alex.fd(laenge)
alex.rt(90)
alex.fd(laenge)
alex.rt(90)
alex.fd(laenge)
alex.rt(90)
turtle.exitonclick()
```

IDEE

- Wir (be)schreiben einmal, wie man ein Quadrat zeichnet und nutzen das dann 10-mal!

AUSLAGERUNG IN UNTERPROGRAMME

- Ziel: Schreiben eines Unterprogrammes, welches ein Quadrat zeichnet
- Eingaben in des Unterprogramms (benötigte Informationen):
  - Seitenlänge und Startposition

Fachbegriff: Interface

UNTERPROGRAMM QUADRAT

```
import turtle
```

Bereitstellung Grafik

```
bob=turtle.Turtle()
```

```
def quadrat(xstart, ystart, laenge):
    bob.penup()
    bob.goto(xstart, ystart)
    bob.pendown()
    for _ in range(0,4):
        bob.fd(laenge)
        bob.rt(90)
```

Beginn Definition Unterprogramm + Parameter

Programmierung der Funktionalität

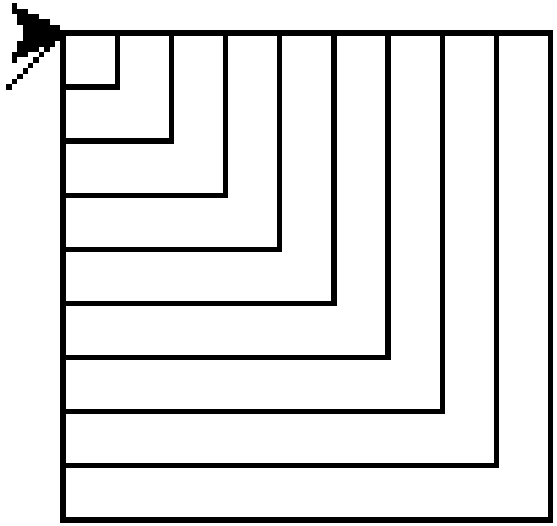
```
quadrat(50,50,100)
```

Aufruf des Unterprogramms, Parameter benötigen konkrete Werte

QUADRACEPTION

```
for i in range(1,10):
    quadrat(10,10,i*10)

turtle.exitonclick()
```



BEISPIEL KOMPLEX

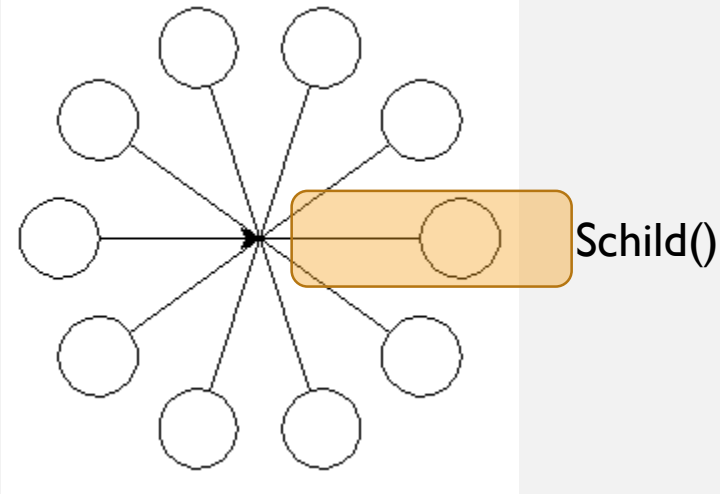
```
import turtle
from random import *

bob=turtle.Turtle() #Objekt erzeugen

def schild():
    bob.fd(80)
    bob.rt(90)
    bob.circle(20)
    bob.lt(90)
    bob.bk(80)

def schildmuehle():
    n=randint(4,20)
    for i in range(n):
        schild()
        bob.rt(360/n)

schildmuehle()
turtle.exitonclick()
```



UNTERPROGRAMME SYNTAX

- Unterprogramme haben immer das folgende Schema:

```
def foo(parameter):
    Anweisungen
    return Rückgabe
```

- Unterprogramme werden immer mit ihrem Namen und konkreten Parametern aufgerufen.
- Sollte die Funktion eine Rückgabe haben, muss das Ergebnis der Funktion in einer Variable gespeichert werden:

```
Ergebnis=foo(parameter)
```

BEISPIEL – UNTERPROGRAMM MIT RÜCKGABE

```
def doppelt(x):
    ergebnis=2*x
    return ergebnis

#Variante 1
wert=doppelt(5)
print(wert)

#Variante 2
for i in range(0,10):
    print(doppelt(i))
```

Definition
- ✓ Name: (doppelt)
- ✓ Parameter: (x)
- ✓ Rückgabe

Aufruf 1:
- ✓ Mit Namen und Parameter
- ✓ Ergebnis wird in Variable gespeichert

Aufruf 2:
- ✓ Mit Namen und Parameter
- ✓ Ergebnis wird direkt ausgegeben

HINWEISE

- Funktionen können verkettet werden
- Beispiel Mathematik: $f(x) = 2 + x, \quad g(x) = x^2$

$$g\big(f(x)\big) = (2 + x)^2$$

- Python:

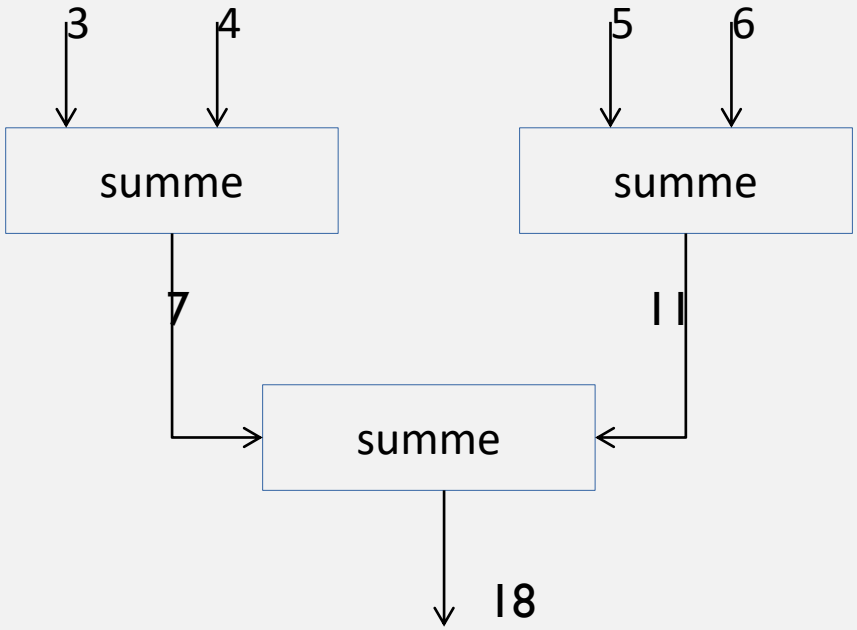
```
print(doppelt(i))
```

- Die Print Funktion wird auf die Doppelt Funktion angewendet

VERKETTUNG

```
def summe(x,y):
    return x+y

verrueckt = summe(summe(3,4), summe(5,6))
print(verrueckt)
```



BEGRIFFE

```
#Definition
def quadrat(xstart, ystart, laenge):


#Aufruf
quadrat(50,50,100)
```

Die in der Funktionsdefinition verwendeten Parameter nennt man formale Parameter.

Beim Aufruf werden diese durch aktuelle Parameter ersetzt (sprich: konkrete Werte).

ÜBUNGEN

Schreibe Unterprogramme, die

- eine Liste zufällig mit Zahlen füllt (als UP)
- die Summe dieser Liste berechnet (als UP)
- den Mittelwert dieser Liste berechnet (als UP, UP Summe nutzen)
- Min und Max der Liste bestimmt (jeweils als UP)

Überlegung gemeinsam: Interface

```
from random import*

def sumListe(liste):
    sum=0
    for ele in liste:
        sum=sum+ele
    return sum

def mittelwert(liste):
    return sumListe(liste)/len(liste)

def FillListe(n):
    list=[]
    for i in range(0,n):
        list.append(randint(0,n))
    return list

def MinMax(liste):
    min=liste[0]
    max=liste[0]
    erg=[]
    for ele in liste:  #Das geht noch einen Schritt schneller, aber die Lesbarkeit
        if ele<min:
            min=ele
        if ele>max:
            max=ele
    return (min,max) #Neu – das nennt man ein Tupel

Liste=FillListe(10)
print(Liste)
print("Summe ist:", sumListe(Liste))
print("Mittelwert ist:", mittelwert(Liste))
print(MinMax(Liste))
```

PARAMETER ÄNDERN

- Funktionen können Parameter übergeben werden
- Den Einfluss der Funktion auf den Parameter hängt vom Objekt ab
  - Veränderliche Objekte: Listen, numerische Datentypen
  - Unveränderliche Objekte: Strings

BEISPIEL I

```
my_list = [1,2,3]

def foo(any_list):
    any_list.append(4)

foo(my_list)

print(my_list)
# Ergebnis: [1,2,3,4]
```

Das Argument kann innerhalb der Funktion geändert werden.
Auch außerhalb der Funktion ist diese Änderung gültig.

BEISPIEL 2

```
my_list = [1,2,3]

def foo(any_list):
    any_list = [1,2,3,4]

foo(my_list)

print(my_list)
# Ergebnis: [1,2,3]
```

Dem übergebenen Parameter wird ein neues Objekt (hier: [1,2,3,4]) zugewiesen.

Das entspricht der Definition einer lokalen Variablen. Any_List ist also lokal.

Die Änderung ist damit nur lokal.

BEISPIEL 3

```
my_string = 'Hello World!'

def foo(any_string):
  any_string.replace('World', 'Python')

foo(my_string)

print(my_string)
# Ergebnis: 'Hello World!'
```

Der übergebene Parameter ist ein String. Dieser gilt als unveränderlich. Es wird daher nur eine Kopie übergeben.

Die Änderung ist daher auch nur lokal.

BEISPIEL 4

```
my_string = 'Hello World!'

def foo(any_string):
  x = any_string.replace('World', 'Python')
  return x

my_string = foo(my_string)

print(my_string)
# Ergebnis: 'Hello Python!'
```

Eine neues Zuweisung zu einer String Variablen wird durch eine neue Zuweisung realisiert.

FALLSTRICKE

- Was ist in folgendem Quellcode übersehen worden?

```
def absoluteValue(x):
  if x < 0:
    return -x
  elif x > 0:
    return x
```

- `Print(absoluteValue(0)) >>> None`
- Tipp: Soll deine Funktion einen Wert zurückgeben, dann stelle sicher, dass zu jeder möglichen Eingabe ein Rückgabewert existiert.
  Murphys Law: Den Fall, den du nicht beachtet hast, wird dein Lehrer in der Klausur testen.