Universität Potsdam
Institut für Informatik

GdP-Rechnerübung

Aufgabenblatt 8

Lernziele (zum Abhaken): Nach diesem Aufgabenblatt sollten Sie...
☐ `for`-Schleifen einsetzen können
☐ die Ausführung von Schleifen mit `break` und `continue` steuern können
☐ Funktionen definieren und aufrufen können

16 for-Schleife vs. while-Schleife

1. Schreiben Sie zwei Skripte, die exakt die folgenden Ausgaben erzeugen;
   einmal sollen sie mit einer `while`- und einmal mit einer `for`-Schleife erzeugt werden.

   Hinweis: Mit `print("Ausgabe", end="")` können Sie den Zeilenumbruch am Ende einer Ausgabe unterdrücken.

   a) `6. Zahl 7. Zahl 8. Zahl 9. Zahl 10. Zahl`
   `Ende`

   b) `20 0 -20 -40 -60 -80`
   `Ende`

   c) `0 Ein 1 Loch 2 ist 3 im 4 Eimer`
   `Ende`

   Hinweis: Benutzen Sie für diese Teilaufgabe eine Liste.

17 Schleifenkontrollen

Auf dem letzten Übungsblatt haben Sie gelernt, wie man mit `while`-Schleifen Anweisungen mehrmals wiederholt ausführen kann. Um die Kontrolle von Schleifen einfacher zu gestalten, gibt es die zwei Kontrollanweisungen `break` *und* `continue`*. Zusätzlich gibt es in Python die leere Platzhalter-Anweisung* `pass`*, die exakt* nichts *macht.*

1. Erschließen Sie sich den Unterschied von `break` und `continue` am nachfolgenden Beispiel:

   Skript 1: `break_continue.py`

```
1 n = 0
2 while n >= 0:
3     print("=== Schleifenblock beginnt ===")
4     n = int(input("Ganze Zahl eingeben: n = "))
5     if n == 0:
6         break
```

```
7       if n % 2 == 0:
8           continue
9       print ("Zahl ist ungerade")
10      print ("=== Schleifenblock endet ===")
11  if n < 0:
12      print("Negative Zahl eingegeben.")
13
14  print("Skript wird beendet.")
```

a) Wohin springt die Ausführung, wenn `continue` ausgeführt wird?

b) Wohin springt die Ausführung, wenn `break` ausgeführt wird?

c) Was passiert, wenn die Bedingung der `while`-Schleife nicht mehr erfüllt ist?

Die Anweisung `pass` dient als Platzhalter für zukünftigen Code. Sie ist nützlich, um die Struktur des Programms zu erstellen und später Funktionen mit Sinn zu füllen:

```
if x==5:
    pass
else:
    pass
```

Ein vollkommen unnützes Python-Programm:

```
while True:
    pass  # beenden mit Strg+C
```

Hinweis: Hier wird ein sogenanntes „Busy Waiting“ durchgeführt. Es wird permanent die Bedingung der `while`-Schleife überprüft, wodurch das System voll ausgelastet wird. Um die CPU tatsächlich freizugeben, kann man einen Warte-Befehl in den Code einbauen:

```
import time
while True:
    time.sleep(2) # Ausführung wird für 2 s unterbrochen
```

18 Funktionen

Funktionen sind ein bequemer Weg, den Quelltext in übersichtliche Teile zu gliedern. Kleine Aufgaben lassen sich einfacher durchdenken und programmieren. Außerdem kann man auf diese Weise einzelne Codeblöcke wiederverwerten, ohne sie erneut kopieren zu müssen – ein einfacher Funktionsaufruf reicht.

```
def <Name>([<Parameterliste>]):
    <Anweisungsfolge>
    return <Rückgabewert>
```

1. Erschließen Sie sich anhand folgender Beispiele[1], wie Funktionen arbeiten:

```
def Hallo():
    print("Hallo Welt!")

Hallo()
Hallo()
```

```
def MehrHallo(anzahl):
    for i in range(0, anzahl):
        print("Hallo Welt!")

MehrHallo(2)
```

```
def mult(a, b):
    return a * b

x = mult(2, 3)
print x, mult(3, 4)
```

2. Schreiben Sie eigene Prozeduren und Funktionen!

   a) Die Prozedur `print_range` erhält zwei Zahlen als Eingabeparameter. Sie gibt den Bereich zwischen diesen Zahlen aus.

   Beispiel: *`print_range(1, 5)` schreibt die Ausgabe: `1 2 3 4 5`*

   b) Die Funktion `list_multiply` erhält eine Liste als Eingabeparameter. Sie multipliziert jeden Wert der Liste mit seinem Index.

   Beispiel: *`list_multiply([1, 2, 3])` gibt zurück: `[0, 2, 6]`*

   Testen Sie Ihre Prozedur und Ihre Funktion in einem Skript. Rufen Sie dabei `list_multiply` mit der Liste `list=[["a"], 1, "a", ['b'], 2, 'b']` auf. Welchen Rückgabewert liefert die Funktion?

[1] mehr Beispiele gibt es auf `https://de.wikibooks.org/wiki/Python_unter_Linux:_Funktionen`

3. Gegeben ist das folgende Skript:

Skript 2: `inc.py`

```
1 def inc(x):
2     x = x + 1
3
4 a = 10
5 inc(a)
6 print(a)
```

a) Führen Sie das Skript aus und überprüfen Sie die Ausgabe!

b) Können Sie innerhalb der Funktion `inc` auf die Variable `a` zugreifen? Überprüfen Sie Ihre Vermutung mit der `print()`-Funktion.

c) Ändert sich das Verhalten durch das Umbenennen von `x` zu `a`?

Was vermuten Sie, warum das so ist?

d) Geben Sie an, wie die Funktion modizifiert werden muss, damit das Skript 11 ausgibt.

4. Schreiben Sie eine zweistellige Funktion, welche Ihnen die Jahreszeit zu einem Datum ausgibt. Benutzen Sie dabei **keine** verschachtelten Kontrollstrukturen! Die Funktion ist wie folgt spezifiziert:

   - Name: `season`
   - Parameter: Tag und Monat als Zahl
   - Rückgabewert: Jahreszeit als Zahl

Rückgabewert	Jahreszeit	Zeitspanne
0	Winter	21.12-19.03.
1	Frühling	20.03-20.06.
2	Sommer	21.06-21.09.
3	Herbst	22.09-20.12.

   Sollte der Benutzer Werte eingeben, die keinen Tag bzw. Monat spezifizieren, dann soll die Funktion eine aussagekräftige Fehlermeldung ausgeben.

Zusatzaufgabe

1. Modifizieren Sie die Funktion aus 4 so, dass sie das Python-Modul `datetime` benutzt. Das Modul enthält den Datentyp `date`, der ein Datum beschreibt. Importieren Sie dafür `date` aus `datetime` (siehe Vorlesung). Den Datentyp können Sie folgendermaßen verwenden:

```
heute = date.today()
silvester = date(2021, 12, 31)
print(silvester.month) // 12
print(heute.day) // z.B. 6
type(heute.year) // <class 'int'>
```

   Die Funktion `season` soll nun also statt Tag und Monat als Zahl einen einzigen Eingabeparameter des Typs `date` erhalten.