

Programmierung in Python

Univ.-Prof. Dr. Martin Hepp, Universität der Bundeswehr München

Einheit 3: Objektorientierte Programmierung in Python

Version: 2019-12-10

<http://www.ebusiness-unibw.org/wiki/Teaching/PIP>

1 Funktionen und Modularisierung

1.1 Motivation

- Programmteile, die oft wiederholt werden, sollten nicht mehrfach vorhanden sein.
- Stattdessen sollte man sie über einen Namen aufrufen können.
- Einen Programmteil, den man über seinen Namen aufrufen kann, nennt man in der Programmierung **Funktion** oder **Methode**.
- Programme werden dadurch kürzer und übersichtlicher.

Beispiel: Im Folgenden wird zwei Mal 'Hallo UniBwM' ausgegeben:

```
In [1]: print('Hallo UniBwM')  
        print('Hallo UniBwM')
```

```
Hallo UniBwM  
Hallo UniBwM
```

Besser wäre es, wenn man diese Funktion ein Mal definieren und dann bei Bedarf aufrufen könnte.

1.2 Definition von Funktionen

Funktionen werden in Python mit dem Schlüsselwort `def` definiert [siehe auch [Python 3 Reference](#)].

```
In [2]: def say_hello():
        print('Hallo UniBwM')
```

Anschließend kann man sie jederzeit über ihren Namen aufrufen:

```
In [3]: say_hello()
        say_hello()
```

```
Hallo UniBwM
Hallo UniBwM
```

Hier spart man zwar nicht wirklich viel an Programmlänge. Aber

Univ.-Prof. Dr. Martin Hepp, Martin.Hepp@unibw.de wenn man den Begrüßungstext ändern möchte, muss man das nur

1.3 Übergabeparameter

Eine Funktion kann so definiert werden, dass man ihr Werte als Parameter übergibt, die dann das Verhalten der Funktion verändern (vgl. [Python 3 Reference](#)).

Dazu definiert man in runden Klammern eine Liste von Namen, über die der jeweils übergebene Wert **innerhalb der Funktion** verfügbar ist.

Beispiel:

```
In [4]: def say_text(text):
        print(text)
```

```
In [5]: say_text('Hallo UniBwM')
```

Natürlich können auch mehrere Parameter übergeben werden:

```
In [6]: def multipliziere(wert_1, wert_2):
        """Multipliziert <wert_1> mit <wert_2>
        ergebnis = wert_1 * wert_2
        print(ergebnis)
```

```
In [7]: # Aufruf mit Werten
        multipliziere(3, 5)
```

15

```
In [8]: # Aufruf mit Variablen
        a = 3
        b = 5
        multipliziere(a, b)
```

15

1.3.1 Arten von Parametern

Beim Aufruf müssen die übergebenen Parameter zu den Parametern in der Definition passen. Die Parameter einer Funktion oder Methode nennt man auch Signatur.

Hier gibt es mehrere Möglichkeiten.

1.3.1.1 Positional Arguments

Parameter können einfach über ihre Reihenfolge bestimmt sein (wie im obigen Beispiel). In den folgenden Beispielen ist der Wert, der beim Aufruf der Funktion `funktion` an erster Stelle übergeben wird, innerhalb der Funktion über den Namen `parameter_1` erreichbar und der Wert, der an zweiter Stelle übergeben wird, über den Namen `parameter_2`:

```
In [9]: def funktion(parameter_1, parameter_2):
        print(parameter_1)
        print(parameter_2)
```

```
        funktion(1, 5)
```

```
1
```

```
5
```

```
In [10]: funktion(5, 1)
```

```
5
```

```
1
```


Die Reihenfolge bestimmt also die Zuordnung:

```
In [11]: def teile(dividend, divisor):
          print(dividend / divisor)

          teile(15, 3)

          5.0
```

```
In [12]: teile(3, 5)

          0.6
```

1.3.1.2 Keyword Arguments

Alternativ kann man auch Namen für Parameter vorgeben, die dann verwendet werden müssen. Dies erlaubt auch Default-Werte.

```
In [13]: def funktion(parameter_name='UniBwM') :  
         print(parameter_name)
```

```
In [14]: # Aufruf mit Parameter  
funktion(parameter_name='hallo')  
  
hallo
```

```
In [15]: # Variable text wird als Wert überg  
text = 'TUM'  
funktion(parameter_name=text)  
  
TUM
```

```
In [16]: # Wenn der Parameter fehlt, wird der Default-Wert verwendet:  
funktion()  
  
UniBwM
```

Keyword Arguments können auch über ihre Position angesprochen werden:

```
In [17]: funktion('Guten Tag')
```

```
Guten Tag
```

1.3.1.3 Variable Anzahl an Positional Arguments (nicht klausurrelevant)

Es ist auch möglich, eine flexible Anzahl an Positional Arguments zuzulassen. Dazu fügt man einen Parameternamen mit vorgestelltem Sternchen ein, wie z.B. `*weitere_parameter`. Dann sind alle weiteren übergebenen Parameter im Inneren der Funktion über ein Tupel mit dem Namen `weitere_parameter` ansprechbar.

Die folgende Funktion erfordert mindestens ein Parameter als Nachnamen und 0..n weitere Parameter als Vornamen.

```
In [18]: def schreibe_namen(nachname, *vornamen):
          for vorname in vornamen:
              print(vorname, end=' ') # end=' ' unterdrückt den Zeilenumbruch
          print(nachname)
```

```
Univ.-Prof. Dr. M In [19]: schreibe_namen('Müller')
```

1.3.1.4 Variable Anzahl an Keyword Arguments (nicht klausurrelevant)

Es ist ebenso möglich, eine flexible Anzahl an Keyword Arguments zuzulassen. Dazu fügt man einen Parameternamen mit vorgestelltem Doppelstern ein, wie z.B. `**weitere_werte`. Dann sind alle weiteren übergebenen Parameter im Inneren der Funktion über ein Dictionary mit dem Namen `weitere_werte` ansprechbar.

Die folgende Funktion erfordert mindestens ein Parameter als Kundennummer und 0..n weitere Paare aus Keyword und Wert als zusätzliche Parameter.

```
In [23]: def drucken_kundendaten(kundennummer, **eigenschaften):  
          print('Kundennummer:', kundennummer)  
          for key in eigenschaften:  
              print(key, '=', eigenschaften[key])
```

1.3.1.5 Weitere Informationen zu Parametern in Funktionen (nicht klausurrelevant)

Siehe <https://docs.python.org/3.7/tutorial/controlflow.html#more-on-defining-functions>.

1.4 Rückgabewerte

Eine Funktion kann einen Wert als Ergebnis zurückliefern. Dazu dient das Schlüsselwort `return`.

Dann muss man den Aufruf der Funktion einer Variable zuweisen, ausgeben oder auf andere Weise in einem Ausdruck verwenden:

```
In [25]: def verdopple(parameter_1):  
         ergebnis = parameter_1 * 2  
         return ergebnis  
  
         print(verdopple(10))  
         mein_wert = verdopple(10)  
         print(mein_wert)
```

20

20

Der Aufruf einer Funktion mit Rückgabewert kann wie jeder andere Wert oder Ausdruck verwendet werden.

```
In [26]: print(verdopple(verdopple(4)))
```

16

```
In [27]: print(verdopple(10) * 1.19)
```

23.799999999999997

Wenn man mehrere Werte zurückliefern möchte, muss man einen Datentyp verwenden, der Unterelemente enthält. Beispiele:

- Tupel
- Dictionary
- Benutzerdefinierte Objekte

```
In [28]: def zwei_werte(parameter_1):
          wert_1 = parameter_1 * 2
          wert_2 = parameter_1 * 3
          return (wert_1, wert_2)

          ergebnis = zwei_werte(4)
          print(ergebnis)
          a, b = ergebnis
          print(a, b)
```

Übungsaufgabe: Schreiben Sie eine Funktion, die die Quersumme einer als Parameter übergebenen Ganzzahl zurückliefert.

```
In [29]: def quersumme(zahl):
          zahl = str(zahl)
          ergebnis = 0
          for ziffer in zahl:
              ergebnis = ergebnis + int(ziffer)
          return ergebnis

          print(quersumme(426))
```

12

1.5 Dokumentation von Funktionen (nicht klausurrelevant)

Jede Funktion sollte eine kurze Beschreibung und Angaben über Aufruf- und Rückgabeparameter enthalten.

Eine genaue Beschreibung dieser 'docstrings' finden Sie unter den folgenden Links:

- <https://docs.python.org/3/tutorial/controlflow.html#documentation-strings>
- <https://www.python.org/dev/peps/pep-0257/>
- <http://google.github.io/styleguide/pyguide.html#383-functions-and-methods>

Beispiel:

```
In [30]: def quersumme(zahl):
        """Berechnet die Quersumme einer Ganzzahl.

        Diese Funktion addiert die Ziffernwerte der übergebenen Ganzzahl.

        Args:
            zahl: Die zu verwendende Ganzzahl.

        Returns:
            Die Quersumme als Ganzzahl.
        """
        zahl = str(zahl)
        ergebnis = 0
        for ziffer in zahl:
            ergebnis = ergebnis + int(ziffer)
        return ergebnis
```

2 Bibliotheken und Import von Modulen (nicht klausurrelevant)

In Python können Module mit vordefinierten Funktionen und Klassen (s.u.) bei Bedarf zum eigenen Programm hinzugefügt werden [[Python 3 Reference](#)].

2.1 Import von Modulen mit `import`

Durch die Anweisung `import <modulname>` wird das Modul `<modulname>` geladen. Danach können alle Namen aus diesem Modul über `<modulname>.<lokaler_name>` angesprochen werden.

Beispiel:

```
In [31]: import math

# Im Modul math gibt es einen Wert pi
print(math.pi)
```

3.141592653589793

```
In [32]: # Und eine Funktion ceil(x), die die kleinste Ganzzahl zurückliefert, die größer is
print(math.ceil(3.14))
```

4

Für weitere Informationen, siehe [hier](#). Die vollständigen technischen Details sind etwas kompliziert und [hier](#) beschrieben.

2.2 Standard-Bibliothek von Python

Python enthält eine sehr reichhaltige Bibliothek an vordefinierten Funktionen und Klassen. Für eine vollständige Liste, siehe [The Python Standard Library](#).

2.3 Vorinstallierte Bibliotheken in Anaconda

Die Anaconda-Distribution enthält viele weitere, vorinstallierte Module; diese variieren zum Teile je nach Betriebssystem / Plattform. Eine Liste ist [hier](#) verfügbar.

2.4 Installation zusätzlicher Bibliotheken

Dieses Thema wird im Rahmen der Vorlesung nicht behandelt und hier nur der Vollständigkeit halber erwähnt. Für eine Liste extern verfügbarer Module, siehe <https://pypi.org/>.

3 Objektorientierung als Programmierparadigma

3.1 Motivation

3.2 Klassen

3.3 Instanzen

3.4 Methoden

3.5 Beispiel

4 Definition einer Klasse (nicht klausurrelevant)

<https://docs.python.org/3/tutorial/classes.html>

4.1 Grundgerüst

- Klasse
- Parameter
- DOCSTRING
- Konstruktor-Methode
- Klassenmethoden
- Klassenvariablen
- Instanzenvariablen

4.2 Variablen auf Instanzebene

Dieses Thema wird im HT 2019 nicht behandelt.

4.3 Konstruktor-Methode

Dieses Thema wird im HT 2019 nicht behandelt.

4.4 Methoden auf Klassenebene

Dieses Thema wird im HT 2019 nicht behandelt.

5 Quellenangaben und weiterführende Literatur

[Pyt2019] Python Software Foundation. Python 3.8.0 Documentation. <https://docs.python.org/3/>.

Vielen Dank!

<http://www.ebusiness-unibw.org/wiki/Teaching/PIP>