LU09a - Funktionen

Funktionen sind kleine, eigenständige Programmteile. Sie haben ihre eigenen Variablen und kommunizieren mit ihrer Umgebung über eine Schnittstelle.

Sinn von Funktionen

Wieso soll man überhaupt ein Programm in Funktionen aufteilen? Zunächst bedeutet das schreiben der Funktion scheinbar einen Mehraufwand. Der Grund wird offensichtlich, wenn man ein wenig weiter schaut.

Wiederhole Dich nicht (DRY)

Das DRY-Prinzip lautet: Don't repeat yourself! Jedesmal wenn Sie den gleichen Code mehrfach schreiben, erhöht sich das Risiko von Fehlern. Angenommen Sie haben in Ihrem Programm den gleichen Codeteil 5x geschrieben. Nun stellen Sie fest, dass ein Fehler in diesem Codeteil ist oder Sie wollen etwas ergänzen. So ist die Gefahr gross, dass Sie die Korrektur 4x vornehmen und die letzte Stelle vergessen.

Daher wird alles, was Sie mehrfach benötigen, in eine Funktion gepackt.

Teilaufgaben erhöhen die Übersicht

Es ist einfacher ein Programm zu verstehen, wenn es in viele, klar definierte Teilaufgaben unterteilt ist. Ein Sourcecode mit mehreren hundert Zeilen an einem Stück, ist extrem schwer zu lesen.

Wartung

Nachdem ein Programm geschrieben wurde, ist es oftmals jahrelang im Einsatz. Während dieser Zeit müssen Fehler korrigiert und weitere Funktionen eingebaut werden. Durch die Abgrenzung von Teilaufgaben wird das Programm übersichtlicher. Es ist auch Jahre später viel einfacher, Änderungen vorzunehmen.

Recycling

Funktionen können sehr einfach wiederverwendet werden. Sie müssen also nicht jedes mal das ganze Programm neu erfinden. Vielmehr können sie Teile aus bestehenden Programmen übernehmen.

Beispielsweise stehen in Python Dutzende von Bibliotheken mit vorgefertigten Funktionen zur Verfügung. Diese Funktiomnen hat ein anderer Softwareentwickler erstellt und wir können Sie nun nutzen. Damit ersparen wir uns viele Stunden Arbeit.

Vorgehen

Gedankenspiel: Auftrag einem Mitarbeiter erteilen

Sie wollen einem Kollegen den Auftrag erteilen, ihnen einen Kaffee aus dem Automaten zu holen. Dieser Kollege weiss, wo der Automat steht und wie man den Kaffeeautomaten bedient.

- Was müssen sie dem Kollegen alles mitteilen bzw. mitgeben?
- Was wollen sie erhalten, wenn der Auftrag erledigt ist?

Lösungsidee

Was muss ich dem Kollegen mitteilen bzw. mitgeben?

- Ob ich einen Kaffee oder einen Espresso möchte.
- Ob ich Rahm und/oder Zucker möchte.
- Das Geld für den Automaten.
- Meine Kaffeetasse.

Was will ich erhalten?

- Meine Kaffeetasse mit dem gewünschten Kaffee drin.

Fassen wir das in Pseudocode zusammen könnte der Auftrag in etwa so aussehen:

```
money = 4.50
size = "Espresso"
cream = False
sugar = True
cup = Cup()

coffee =
colleague.bring_coffee(money, size,
cream, sugar, cup)
```

Funktion realisieren

Nun gehen Sie daran, die Funktion zu realisieren. Dabei empfiehlt es sich, gezielt vorzugehen:

1. Funktionsblock definieren

2. Dokumentationszeichenkette (docstring)
3. Logik programmieren

1. Funktionsblock definieren

Bezeichner der Funktion definieren

Als ersten Schritt überlegen Sie sich, welche Teilaufgabe diese Funktion erledigen soll. Diese Teilaufgabe muss sich in einem kurzen Satz prägnant umschreiben lassen. Unklare oder schwammig definierte Teilaufgaben führen in der Regel zu schlechten Funktionen.

- Vermeiden Sie Füllwörter wie Ding, etwas, machen, ...
- Eine Methode hat genau eine Aufgabe ⇒ Beschreibungen mit **und** bzw. **oder** vermeiden.

Aus der Definition der Aufgabe lässt sich auch ein sinnvoller Name für die Funktion ableiten.

Beispiele:

- Beschreibung: Die Methode gibt einen String an der aktuellen Cursorposition aus.
  - Name: `put_string`
- Beschreibung: Die Methode prüft, ob die Eingabe des Benutzers eine Zahl ist.
  - Name: `is_number`

Aus dem Namen der Funktion soll sich also auch der Zweck der Funktion herleiten lassen (Sprechend sein). Funktionsnamen sind klein geschrieben. Handelt es sich um mehr als ein Wort, so ist die snake_case_schreibweise angebracht. Jede Funktionsdefinition beginnt mit dem Wort `def` um Python anzuzeigen, dass eine neue Funktion definiert wird.

```
def put_string():
    ...
```

Parameter festlegen

Zur Schnittstelle gehören auch die Parameter/Argumente, welche der Aufrufer an die Funktion weiter gibt. Jeder Parameter benötigt einen einmaligen Bezeichner

```
def check_login(username, password): # Korrekte Bezeichner
    ...

def check_name_identical(name, name): # führt zu einem Fehler, da die
Parameter identische Bezeichner haben
```

Zu jedem Paramter in der Schnittstelle wird innerhalb der Funktion automatisch eine lokale Variable deklariert. Im Beispiel `check_login` stehen also innerhalb der Methode die Referenzvariablen `username` und `password` zur Verfügung.

Beim Aufruf der Funktion gibt der Aufrufer die entsprechenden Werte für diese Variablen mit. Diese Werte werden automatisch in die entsprechenden lokalen Variablen gespeichert.

Das ist die Codezeile mit `def`, dem Bezeichner und eventuellen `(Parameter/n)` abgeschlossen mit einem `:`:

```
def is_number(number):
    ...
```

Alles was jetzt in die Funktion gehört muss identisch eingerückt sein. Normalerweise werden dafür 4 Leerzeichen verwendet.

2. Dokumentationszeichenkette (docstring)

Nun kommt der Kommentarblock mit den Angaben zur Funktion. In vielen Entwicklungswerkzeugen wird ein Teil dieses Kommentarblocks automatisch erzeugt. In PyCharm musst Du dafür ein integriertes Werkzeug aktiveren. Erstellen Sie dazu drei Anführungzeichen **„“„** unterhalb der Funktionsdeklaration und drücken Sie `enter`

```
def is_big(number):
    """

    :param number:
    :return:
    """

}
```

Ergänzen Sie nun den generierten Docstring mit einer aussagekräftigen Beschreibung und der Beschreibung der Parameter und des Return-Wertes. Falls sie keinen Return-Wert geplant haben, schreiben Sie `None` hinter `:return:`

```
def is_big(number):
    """
    Checks if a given number is a big number
    :param number (int): the number to check
    :return (bool): True or False depending on the size of the number
    """


def say_hello():
    """
    Just says 'Hello'
    :return: None
    """
```

```
def multiply(factor1, factor2):
    """
    Multiplies the two numbers factor1 * factor2
    :param factor1 (float): the first number
    :param factor2 (float): the second number
    :return product (float): product of the multiplication
    """
```

Können Sie die `docstrings` nicht automatisch erzeugen,
folgen Sie dieser Anleitung.

Weitere Ergänzungen zu den Kommentaren finden Sie in PEP
257 oder in den Coding-Guidelines des BZZ

3. Logik programmieren

Zuletzt programmieren Sie die Logik der Funktion. Betrachten Sie jede Methode wie ein eigenständiges kleines Programm, mit:

- Variablen
- Anweisungen
- Returnwert

```
def is_big(number):
    """
    Checks if a given number is a big number
    :param number (int): the number to check
    :return (bool): True or False depending on the size of the number
    """
    threshold = 1000000

    if number > threshold:
        return True
    else:
        return False


def say_hello():
    """
    Just says 'Hello'
    :return: None
    """
    print('Hello')


def multiply(factor1, factor2):
    """
    Multiplies the two numbers factor1 * factor2
```

```
    :param factor1 (float): the first number
    :param factor2 (float): the second number
    :return product (float): product of the multiplication
    """
    product = factor1 * factor2
    return product
```

Methoden aufrufen

Um eine Funktion aufzurufen, geben wir den Namen der Funktion und runde Klammern an:

- `main()` Ruft eine Funktion ohne Parameter auf.
- `is_big(123)` Ruft eine Funktion mit einem Parameter auf.

Beispiel:

```
def is_big(number):
    """
    Checks if a given number is a big number
    :param number (int): the number to check
    :return (bool): True or False depends on
the size of the number
    """
    threshold = 1000000

    if number > threshold:
        return True
    else:
        return False

def main():
    is_it = is_big(123)
    print(is_it)

if __name__ == '__main__':
    main()
```

```
False
```

Hinweis: In Python müssen alle Funktionsdefinitionen **vor** dem Block `if __name__ == '__main__':` stehen. Andernfalls erhalten wir einen Fehler. Zum Beispiel:

```
def main():
```

```
    is_it = is_big(123)
    print(is_it)


if __name__ == '__main__':
    main()


def is_big(number):
    """
    Checks if a given number is a big
number
    :param number (int): the number to
check
    :return (bool): True or False
depends on the size of the number
    """
    threshold = 1000000

    if number > threshold:
        return True
    else:
        return False
```

NameError: name 'is_big' is not defined.

M319-F2G, M319-F2F, M319-F2E

© Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/de/modul/m319/learningunits/lu09/funktionen?rev=1763028104

Last update: **2025/11/13 11:01**