Programmierung mit Python für Einsteiger

Kapitel 8 - Fallbeispiel Zeiterfassung

Autor: Dr. Christian Heckler

Fallbeispiel

In der folgenden Sequenz von Übungen soll in mehreren kleinen Schritten ein Fallbeispiel für die Zeiterfassung programmiert werden.

Problemstellung: Ein Mitarbeiter arbeitet am Tag für verschiedene Projekte / Themen, z.B.

- Pythonkurs vorbereiten
- Programmierung einer Bankanwendung
- Betrieb einer Bankanwendung

Am Ende des Tages möchte der Mitarbeier wissen, wie lange er für welches Projekt gearbeitet hat.

Mögliche Erweiterung: Über einen Zeitraum X möchte der Mitarbeiter eine Auswertung über die Arbeitszeiten.

Anwendung Zeiterfassung

- Auf „Knopfdruck“ erfasst der Mitarbeiter, wenn er die Arbeit an einem Projekt startet bzw. beendet.
- Diese „Ereignisse“ werden in einer Datei gespeichert. Eine Zeile in der Datei repräsentiert dabei ein Ereignis:
  - Start oder Stop
  - Projekt
  - Uhrzeit
- Beispiel:
  - Start, Vorbereitung Python-Kurs, 8:15
  - Ende, Vorbereitung Python-Kurs, 10:05
- Anhand dieser Datei kann dann die Zeit berechnet werden, die für das Projekt aufgewendet wurde (10:05 minus 8:15 = 1:50).

Datum und Uhrzeit

- Datentypen für Datum und Uhrzeit gehören nicht zu den eingebauten Python-Datentypen.
- Es gibt ein entsprechendes Modul (das evtl. am Ende vorgestellt wird).
- Wir repräsentieren zunächst die Uhrzeit durch zwei ganze Zahlen:
  - Stunde
  - Minute
- Dadurch sind nur Auswertungen innerhalb eines Tages möglich.
- Später ist es leicht möglich (durch OO), diese Repräsentation der Uhrzeit durch einen „Zeitstempel“ aus dem entsprechenden Python-Modul zu ersetzen.

Übung 1.1: Differenz zweier Uhrzeiten

- Schreiben Sie ein Programm:
  - Definition von 4 ganzzahligen Variablen:
    - stunde_1, minute_1
    - stunde_2, minute_2
  - Die Differenz soll in der Form Stunde, Minute ausgegeben werden
- Beispiel:
  - stunde_1=8, minute_1=15, stunde_2=10, minute_2=5
  - Ausgabe: 1:50
- Hinweise:
  - Differenz über Anzahl der Minuten seit Mitternacht
  - Umrechnung in Stunde, Minute über ganzzahlige Div

Übung 1.2: Datenstrukturen

- Überlegen Sie sich eine geeignete Datenstruktur zum Speichern mehrerer „Ereignisse“.
  - Datenstruktur für Ereignisse?
  - Datenstruktur für mehrere Ereignisse?

Übung 2.1: Differenz über `if`-Anweisung

- Wie Übung 1.1: Differenzbildung von zwei Uhrzeiten, die jeweils durch zwei ganze Zahlen repräsentiert werden (Stunde, Minute).
- Diesmal ohne Verwendung von Multiplikation und Division.
- Nur die Operationen `+` und `-` sind erlaubt und die `if`-Anweisung

❓ Übung 2.2: Schleife über Ereignisse

- Gegeben sei eine Liste von Ereignissen modelliert wie in Übung 1.2 vorgeschlagen.
- Annahmen:
  - Die Einträge in der Liste sind nur für einen Tag.
  - Die Einträge sind sortiert nach der Zeit.
  - Start- und Stop-Ereignisse folgen korrekt aufeinander
- Schreiben Sie ein Programm, das über die Liste schleift und die Gesamtarbeitszeit berechnet und ausgibt (unabhängig vom Projekt).

Übung 2.2: Erweiterung 1

- Wenn auf ein Start-Ereignis ein weiteres Start-Ereignis kommt, geben Sie eine Fehlermeldung aus und überspringen Sie dieses (via `continue`).
- Folgt auf ein Start-Ereignis ein Ereignis eines anderen Projekts, geben Sie eine Fehlermeldung aus und brechen Sie die Schleife ab (`break`).

Übung 2.2: Erweiterung 2

- Berechnen Sie nun die Arbeitszeit pro Projekt. Dabei kann wieder von der ursprünglichen Annahme ausgegangen werden, dass die Ereignisse korrekt aufeinander folgen.
- Idee: In einer ersten Schleife könnte man sich die Ereignisse, die jeweils zu einem Projekt gehören, in einem Dictionary (z.B. projekt_ereignis_list_dict) speichern:
  - Schlüssel: Projektname
  - Eintrag: Liste von Ereignissen für dieses Projekt
- Dazu: Prüfen, ob im Dictionary schon ein Schlüssel für das Projekt vorhanden ist:
  - `if projekt not in projekt_ereignis_list_dict:`
    - Ja: Liste nehmen und erweitern: `l.append(ereignis)`
    - Nein: Neue Liste ins Dict einfügen.

Übung 2.2: Erweiterung 2 - Fortsetzung

- Zweiter Schritt: Über Einträge des Dictionaries schleifen:
  - `for projekt, ereignis_liste in projekt_ereignis_list_dict.items()`
  - Für jeden Eintrag die Arbeitszeit berechnen analog Teil 1

Übung 2.3: Ereignisse aus Datei lesen

- Schreiben Sie ein Programm, das Ereignisse aus einer Datei liest und eine Liste erstellt:
  - Wie in den Übungen vorher wird ein Ereignis dabei durch ein Tupel repräsentiert:
    - `(Projekt, Projekttyp, (Startstunde, Startminute))`
  - Die Ereignisse sind in der Datei folgendermaßen zeilenweise gespeichert:
    - `Python-Kurs, Start, 8, 15`
  - Hinweis:
    - zeile = “Python-Kurs, Start, 8, 15“
    - liste_bestandteile = zeile.split(“,“)
    - Dann hat die Liste 4 Elemente

Übung 3.1: Funktionen für die Zeiterfassung

- Überlegen Sie sich, welche Teile der bisherigen Übungen in Funktionen geschrieben werden könnten.
- Erinnerung: bisherige Aufgaben:
  - Berechnung der Zeitdifferenz zweier Tupel `(Stunde, Minute)`.
  - Für eine Liste von Ereignissen die Gesamtarbeitszeit berechnen.
  - Für eine Liste von Ereignissen: Pro Projekt die Arbeitszeit berechnen.
  - Liste von Ereignissen aus einer Datei lesen.
- Erstellen Sie ein Programm, das eine Liste von Ereignissen aus einer Datei liest und die Arbeitszeit pro Projekt berechnet. Benutzen Sie dazu Funktionen.

Übung 3.2: Kommandozeilenparameter

- Schreiben Sie ein Programm `start.py` mit drei Kommandozeilenparametern:
  - Projekt
  - Startstunde
  - Startminute
- Beim Aufruf soll ein entsprechendes Ereignis in die Datei `ereignisliste.txt` geschrieben werden.
- Beispielaufruf:
  - `python start.py python-kurs 10 30`
- Erzeugte Zeile: `python-kurs, Start, 10, 30`

Übung 3.2: Bemerkung

In einer realistischen Anwendung

- würde man dafür einen Knopf in einer graphischen Oberfläche programmieren,
- würde die Zeit automatisch aus der Systemzeit ermittelt und in die Datei eingetragen,
- und würde es sich um eine komplette Zeitangabe handeln (inklusive Tag, Monat, Jahr).

❓ Übung 3.3: Listensortierung

- Ein „Ereignis“ sei ein Tupel der folgenden Art:
  - `(Projektname, Typ, (Stunde, Minute))`
- Beispiel_
  - `(“Python-Kurs“, “Start“, (10,20))`
- Schreiben Sie ein Programm, das eine Liste von Ereignissen nach der Uhrzeit sortiert.

Übung 3.4: Module

- Die Funktionen für die Zeiterfassung seien nun in einem Modul `zeiterfassung.py` zusammengefasst.
- Schreiben Sie ein Programm `zeiterfassungtest.py`, das mit Hilfe der Funktionen aus dem Modul eine Liste von Ereignissen einliest und die Arbeitszeit pro Projekt ausgibt.

Übung 3.5: Ausnahmen

- Gegeben seien die Funktionen:
  - `berechne_arbeitszeit`: Berechnet die Arbeitszeit für eine Liste von Ereignissen für ein Projekt
  - `berechne_arbeitszeit_pro_projekt`: Bekommt eine Liste von Ereignisse für mehrere Projekte. Berechnet pro Projekt die Arbeitszeit und schreibt diese in ein Dictionary.
- Ändern Sie die erste Funktion so, dass eine Ausnahme geworfen wird, wenn die Ereignisse inkonsistent sind, also wenn z.B. auf ein Ende-Ereignis noch ein Ende-Ereignis folgt.
- Ändern Sie die zweite Funktion so, dass entsprechende Ausnahmen abgefangen werden. Tritt ein Fehler auf, soll eine Meldung ausgegeben werden. Die Arbeitszeit für das entsprechende Projekt soll auf `-1` gesetzt werden.

Übung 3.5: Zusatzaufgabe

- Der momentane Algorithmus beruht darauf, zuerst Listen mit Ereignissen für die einzelnen Projekte zu erstellen, und dann für jedes Projekt die Arbeitszeit separat zu berechnen.
- Auch nach Einbau der Ausnahmebehandlung kann es sein, dass eine Liste von Ereignissen inkonsistent ist. Warum?
- Schreiben Sie eine Prozedur, die eine Liste von Ereignissen auf diese Inkonsistenzen überprüft.
- Was machen Sie, wenn eine solche Inkonsistenz auftritt?

❓ Übung 4.1: Klassen

- Erstellen Sie eine Klasse `Ereignis` mit den Attributen
  - `projekt` (zunächst vom Typ String mit dem Namen des Proj.)
  - `typ` („Start“ / „Ende“)
  - `zeitstempel` (Tupel aus Stunde, Minute – wie bisher)
- Schreiben Sie eine Funktion, die die Ereignisse aus einer Datei liest und eine Liste von Objekten vom Typ Ereignis erzeugt.

Übung 4.1: Klassen - Zusatzaufgabe

- Erstellen Sie zusätzlich eine Klasse `Projekt` mit den Attributen `name` und `beschreibung`.
- Die Klasse `Ereignis` erhält nun nicht mehr den Projekt-Namen als Attribut, sondern ein Attribut `projekt`, das ein Objekt vom Typ `Projekt` enthält.
- Die Funktion zum Einlesen der Ereignisse aus einer Datei soll entsprechend angepasst werden.
- Dabei kann das Attribut `beschreibung` der Projekt-Objekte erst mal leer bleiben.

Übung 4.1: Klassen - Objektidentität

- Pro Name soll nur ein Projekt-Objekt erzeugt werden. Im Beispiel soll es also zwei Projekt-Objekte geben (Python-Kurs, Java-Projekt).
- In den Ereignissen soll das entsprechende Projekt-Objekt verwendet werden. Ereignisse zum gleichen Projekt sollen also **dasselbe** Projekt-Objekt referenzieren!
- Dazu erstellt man beispielsweise beim Einladen ein Dictionary:
  - Schlüssel: Projekt-Name
  - Wert: Projekt-Objekt
- Beim Laden eines Ereignisses wird geprüft, ob es im Dictionary schon ein Projekt mit dem Namen gibt.
  - ja: nehme dieses Objekt
  - nein: erzeuge ein entsprechendes neues Projekt-Objek

Übung 4.2: Properties

- Die Attribute der Klasse `Ereignis` sollen als „privat“ markiert werden.
- Für das Attribut `projekt_name` sollen erstellt werden:
  - get-Methode
  - set-Methode
- Für das Attribut sollen entsprechende „Properties“ zur Verfügung gestellt werden, so dass im Hauptprogramm der Projektname folgendermaßen ausgegeben werden kann (also ohne expliziten Aufruf der get-Methode):
  - `print("Projekt: ", ereignis.projekt_name)`

Übung 4.3: Magische Methode `str()`

- Erweitern Sie die Klasse Ereignis um die `str()`-Methode.
- Verwenden Sie diese im Hauptprogramm zur Ausgabe.
- Zusatzaufgabe:
  - Implementieren Sie auch die `repr()`-Methode
  - Schreiben Sie die Ereignisse mit Hilfe dieser Methode in eine Datei.
  - Eine solche Datei kann dann eingelesen werden, in dem ein Ereignis-Objekt aus einer Zeile der Datei folgendermaßen erzeugt wird:
    - `ereignis = eval(zeile.strip())`

Übung 4.4: Magische Methode `sub()`

- Implementieren Sie in der Klasse Ereignis die Methode `sub()`. Diese soll die Differenz in Minuten der Zeitstempel der Ereignisse zurückliefern.
- Testen Sie die Methode.
- Die Methode soll eine Ausnahme werfen, wenn
  - die Ereignisse zu unterschiedlichen Projekten gehören oder
  - wenn nicht ein Start-Ereignis von einem Ende-Ereignis subtrahiert wird oder
  - wenn eines der beiden Objekte gar kein Ereignis ist.

Übung 4.5: Sortierung

- Schreiben Sie eine Funktion, die eine Liste von Objekten vom Typ `Ereignis` sortiert.
  - Implementieren Sie dazu in der Klasse Ereignis eine Methode `sortier_kriterium`, die die Anzahl der Sekunden seit Mitternacht zurückgibt.
  - Diese kann als `key` - Funktion der Sortierfunktion benutzt werden (`key = Ereignis.sortier_kriterium`).

Übung 5.1: Ableitung

- Leiten Sie von der Klasse `Ereignis` zwei Unterklassen ab:
  - `StartEreignis`
  - `EndeEreignis`
- Bemerkung: Das ist nur eine Übung. In dem Beispiel hat das keinen Mehrwert

Übung 6.1: Verwendung des Moduls `datetime`

- Bisher besteht die interne Implementierung der Klasse `Ereignis` darin, den Zeitstempel als Tupel zweier Ganzzahlen (Stunde, Minute) zu speichern. Dies hat den Nachteil, dass es unmöglich ist, Ereignisse für mehrere Tage zu speichern.
- Ändern Sie die Implementierung, so dass ein Zeitstempel mit einem Objekt vom Typ `datetime` aus dem Modul `datetime` verwendet wird.
  - Die Differenz zweier Objekte ist vom Typ `timedelta`. Von diesem kann man sich die Anzahl der Sekunden zurückgeben lassen. Die `Sub`-Methode kann also weiterhin die Anzahl der Sekunden zurückgeben.
  - Die Funktion `sortier_kriterium` ist entsprechend zu ändern.