Eine kleine Datenbank
für Adressen und Ähnliches

Autor: Marc 'BlackJack' Rintsch
Kontakt: marc@rintsch.de
Datum: 2007-05-31
Version: 0.1
Status: Entwurf

Inhaltsverzeichnis

1 Vorbemerkung

Dies ist die *etwas* übertrieben ausführliche Antwort auf eine Diskussion im deutschsprachigen Python-Forum über den Entwurf einer kleinen Datenbank als Projekt für Informatikunterricht. Die hier vorliegende Variante soll ohne *objektorientierte Programmierung* (*OOP*) auskommen. Allerdings auch ohne `global`.

2 Entwurf

Bevor es an die Umsetzung geht, müssen wir uns ein paar Gedanken darüber machen, was die Datenbank können soll und mit welchen Datenstrukturen sie modelliert werden soll. Dabei wird davon ausgegangen, dass die grundlegenden Datentypen von Python, wie Zahlen, Zeichenketten, Listen, Tupel und Dictionaries, sowie die Definition von Funktionen bereits bekannt sind, oder nebenher eingeführt und gelernt werden.[1] Bei den Grunddatentypen würde es allerdings Sinn machen sie vorher bekannt zu machen, denn nur so kann man die verwendeten Datentypen im Vergleich zu anderen Möglichkeiten begründen.

Der Entwurf wird bei den Anforderungen *top-down* entwickelt, die Darstellung der Umsetzung erfolgt dann aber *bottom-up*. Das bedeutet die Planung geht vom grossen Ganzen zu den kleineren Details und die Implementierung fängt mit den kleineren Bausteinen an und baut diese zu einer ganzen Lösung zusammen. Auf diese Weise hat man sich über alle Ebenen einmal Gedanken gemacht bevor es losgeht und kann beim Programmieren jederzeit testen, ob das bisherige Ergebnis den Anforderungen entspricht.

2.1 Anforderungen

Da die Datenbank nicht nur für *einen* Einsatzzweck, wie zum Beispiel Adressen, nutzbar sein soll, müssen die Schlüssel für einen Datensatz flexibel gehalten werden.

Das Programm soll folgende Funktionalität bieten:

[1] Im Grunde gehen die Erwartungen an bereits bestehendes Wissen über die Sprache und ihre Konstrukte weiter. Der Quelltext sollte für echte Anfänger etwas einfacher gestaltet werden. Der Schwerpunkt des Dokuments soll vorranging auf dem Entwurf des Programms und der Funktionalität der einzelnen Funktionen liegen.

- anlegen einer Datenbank
- ausgeben der Datenbank und einzelner Datensätze
- eingeben und bearbeiten von Datensätzen
- laden der Datenbank
- speichern der Datenbank
- suchen von Datensätzen
- sortieren der Datensätze

Als Benutzerschnittstelle soll eine sehr schlichte Kommunikation per Textein- und Ausgabe zum Einsatz kommen. Der Code für die Benutzerschnittstelle soll von der Datenhaltung getrennt sein, so dass es möglich ist, diesen Teil durch eine GUI zu ersetzen.

2.2 Datenstrukturen

Unsere kleine Datenbank soll aus Datensätzen bestehen. Ein einzelner Datensatz wiederum besteht aus beliebigen Feldern mit Bezeichnern und Werten und einer ID, die den Datensatz innerhalb der Datenbank eindeutig identifiziert. Im Jargon der „grossen“ Datenbanken entspricht unsere Datenbank einer Tabelle, die Datensätze den Zeilen, und die ID der Row-ID oder Object-ID (OID). Man könnte sich so eine Datenbank als Tabelle wie folgt vorstellen:

Tabelle 1: Einfache Datenbank.

ID	Name	Beruf
0	Erik	Wikinger
1	Brain	Weltherrscher

Wie man an der Darstellung der Tabelle sieht, braucht man die Schlüssel nur einmal für die gesamte Datenbank speichern und bei den Datensätzen nur die Daten in jeder Zeile. Für beides bieten sich Tupel an. Die ID nimmt eine Sonderstellung ein, da sie vom Quelltext der Datenbank verwaltet wird und nicht vom Benutzer des Programms verändert werden kann, weil dann die Gefahr besteht, das die ID nicht mehr eindeutig ist. Ein einzelner Datensatz besteht damit auch aus zwei Teilen: der ID und den Werten. Auch das lässt sich mit einem Tupel modellieren. Damit sähe ein Datensatz so aus:

```
>>> erik = (0, ('Erik', 'Wikinger'))
```

Da in den Anforderungen das Sortieren der Datenbank vorgesehen ist, braucht man einen Container-Datentyp, der die Ordnung der Elemente erhält. Eine Liste bietet sich hier an. Der Datengehalt obiger Tabelle lässt sich also so darstellen:

```
>>> brain = (1, ('Brain', 'Weltherrscher'))
>>> records = [erik, brain]
>>> records
[(0, ('Erik', 'Wikinger')), (1, ('Brain', 'Weltherrscher'))]
```

Bei den Schlüsseln braucht man die ID nicht extra erwähnen, weil sie fester Bestandteil eines jeden Datensatzes ist. Die Tabelle weiter oben liesse sich als Datenstruktur also wie folgt komplett mit Python's Grunddatentypen darstellen:

```
>>> TEST_DB = (('name', 'beruf'),           # Schluessel.
...            [(0, ('Erik', 'Wikinger')),  # Zwei Datensaetze.
```

```
...             (1, ('Brain', 'Weltherrscher'))])
```

3 Umsetzung

Nun folgt die Umsetzung des Entwurfs. Die Funktionen werden in einer Reihenfolge eingeführt, die zuerst die kleinen Teile der Datenbank, wie Werte und Datensätze betreffen, und dann zu den grösseren Strukturen übergeht, bis am Ende der Quelltext für die Benutzerinteraktion hinzu kommt.

Begleitend zum Quelltext gibt es kleine Beispiele für die Anwendung der Funktionen in einer interaktiven Python-Sitzung. Der Anfang wird mit dem Import des Moduls gemacht, gefolgt von der Definition von Konstanten einer Adress-Datenbank, wie sie auch vom Hauptprogramm verwaltet wird:

```
>>> import simpledb as db
>>> ADDRESS_KEYS = ('name', 'strasse', 'ort', 'tel.')
>>> TEST_ENTRY = (42, ('Max Mustermann',
...                    'Mustergasse 42',
...                    'Musterstadt',
...                    '12345'))
>>> ADDRESSES = (ADDRESS_KEYS, [TEST_ENTRY])
```

3.1 Abhängigkeiten

Das Programm benutzt *pickle* zum Sichern der Daten auf Platte und Funktionen aus dem *sys*-Modul.

```
import pickle
import sys
```

3.2 Konstanten

Obwohl die Datenbank selbst flexibel einsetzbar sein soll, beschränkt sich das Beispielprogramm auf Adressen. Deshalb werden die Schlüssel als Konstante definiert.

```
ADDRESS_KEYS = ('name', 'strasse', 'ort', 'tel.')
```

3.3 Datensatz

In diesem Abschnitt werden die Funktionen besprochen, die auf einzelnen Datensätzen bzw. auf deren Werten operieren.

3.3.1 Datensatz in Zeichenkette umwandeln

Ein einzelner Datensatz wird, zum Beispiel zur Anzeige am Bildschirm, in eine Zeichenkette umgewandelt. Die Darstellung ist dabei eine Zeile pro Feld des Datensatzes und in jeder Zeile ist der Schlüssel mit einem Doppelpunkt getrennt vom Wert des Feldes aufgeführt.

```
def record2str(keys, (id_, values)):
    return (('ID: %s\n' % id_)
            + '\n'.join('%s: %s' % (key.capitalize(), value)
                        for key, value in zip(keys, values)))
```

Auf diese Funktion stützt sich später die Umwandlung der gesamten Datenbank in eine Zeichenkette und die Funktion zum Bearbeiten eines einzelnen Datensatzes gibt diesen vorher aus.

```
>>> print db.record2str(ADDRESS_KEYS, TEST_ENTRY)
ID: 42
Name: Max Mustermann
Strasse: Mustergasse 42
Ort: Musterstadt
Tel.: 12345
```

Mögliche Erweiterung wäre eine bündige Formatierung der Werte, so dass entweder die Schlüssel soweit eingerückt sind, dass die Doppelpunkte untereinander stehen, oder entsprechend mit Leerzeichen aufgefüllt werden. Das funktioniert natürlich nur bei Anzeigen mit einer proportionalen Schriftart.

3.3.2 Werte auf „Mindestbelegung“ testen

Wir möchten bei der Eingabe verhindern, dass ein Datensatz zuwenig Informationen enthält. Bei einer Adresse macht es zum Beispiel wenig Sinn, wenn nicht mindestens ein Name und eine Adresse oder Telefonnummer gespeichert werden; von komplett leeren Datensätzen ganz zu schweigen. Felder deren Wert ausschliesslich aus Leerzeichen besteht, werden als leer angesehen.

```
def at_least_filled(values, minimum):
    return sum(1 for value in values if value.strip()) >= minimum
```

Der Testeintrag erfüllt die Bedingung, dass mindestens zwei Felder mit Werten belegt sind:

```
>>> id_, values = TEST_ENTRY
>>> db.at_least_filled(values, 2)
True
```

Ein komplett leerer Eintrag, auch wenn die Werte aus Leerzeichen bestehen, erfüllt die Bedingung nicht:

```
>>> db.at_least_filled(('', '', '', ''), 2)
False
>>> db.at_least_filled((' ', ' ', ' ', ' '), 2)
False
```

Genausowenig wie ein Name. Fügt man einen Wohnort hinzu, dann genügen die Werte der Bedingung:

```
>>> db.at_least_filled(('Erik', '', '', ''), 2)
False
>>> db.at_least_filled(('Erik', '', 'Valhalla', ''), 2)
True
```

Dieser Test ist natürlich ein wenig zu generell. Besser wäre es, wenn man die Felder, die nicht leer bleiben dürfen, explizit angeben kann. Das wäre ein Punkt für eine Erweiterung des Programms.

3.3.3 Suchen in einem Datensatz

Auf dem Test, ob ein Suchwort in einem Datensatz enthalten ist, baut die Suche in der Datenbank auf. Es werden alle Werte ohne Berücksichtigung von Gross- und Kleinschreibung durchsucht.

```
def record_contains((id_, values), keyword):
    keyword = keyword.lower()
    return True in (keyword in value.lower() for value in values)
```

Die Suche nach einem Suchbegriff der nicht in dem Testeintrag vorkommt, und einem der es tut:

```
>>> db.record_contains(TEST_ENTRY, 'test')
False
>>> db.record_contains(TEST_ENTRY, 'max')
True
```

3.4 Datenbank

Nun kommen wir von einzelnen Werten und Datensätzen zu Funktionen, die auf der Datenbank als ganzes operieren.

3.4.1 Datenbank in Zeichenkette umwandeln

Auch eine Datenbank muss zur Anzeige in eine Zeichenkette umgewandelt werden. Dazu werden die Zeichenketten für die einzelnen Datensätze durch eine Überschrift, Trennlinien und einer Meldung über die Gesamtzahl der Datensätze ergänzt.

```
def to_str(keys, records):
    result = ['::: Datensaetze :::']
    result.extend(record2str(keys, record) for record in records)
    length = len(records)
    if length == 1:
        result.append('1 Datensatz.')
    else:
        result.append('%d Datensaetze.' % length)
    return ('\n%s\n' % ('-' * 20)).join(result)
```

Die Testdatenbank sieht so aus:

```
>>> keys, records = TEST_DB
>>> print db.to_str(keys, records)
::: Datensaetze :::
--------------------
ID: 0
Name: Erik
Beruf: Wikinger
--------------------
ID: 1
Name: Brain
Beruf: Weltherrscher
--------------------
2 Datensaetze.
>>> print db.to_str(('ham', 'eggs'), [])
::: Datensaetze :::
--------------------
0 Datensaetze.
```

Nicht ganz so sauber ist hier die Verwendung von konkreten deutschen Texten bei einer Funktion, die eigentlich allgemein nutzbar sein sollte. Das gehört schon in den Bereich der Benutzerschnittstelle.

3.4.2 Unbenutzte ID ermitteln

Wenn man einen neuen Datensatz in die Datenbank einfügen möchte, so braucht dieser eine eindeutige ID, die noch nicht existiert. Die Funktion sucht die grösste ID in den vorhandenen Datensätzen und erhöht den Wert um eins. Falls es keine Datensätze gibt, wird eine 0 zurückgegeben:

```
def get_unused_id(records):
    try:
        return 1 + max(id_ for (id_, values) in records)
    except ValueError:
        return 0
```

In der Testdatenbank sind die IDs 0 und 1 vorhanden. Bei einer leeren Liste ist die erste ID die 0:

```
>>> keys, records = TEST_DB
>>> db.get_unused_id(records)
2
>>> TEST_ENTRY
(42, ('Max Mustermann', 'Mustergasse 42', 'Musterstadt', '12345'))
>>> db.get_unused_id([TEST_ENTRY])
43
>>> db.get_unused_id([])
0
```

3.4.3 Werte in die Datenbank eintragen

Um neue Werte in die Datenbank einzutragen, muss man einen Datensatz mit einer unbenutzen ID und den Werten an die Liste mit den Datensätzen anhängen.

```
def add_values(records, values):
    records.append((get_unused_id(records), values))
```

Als Test wird die Testdatenbank aus Werten neu aufgebaut.:

```
>>> records = list()
>>> db.add_values(records, ('Erik', 'Wiking'))
>>> db.add_values(records, ('Brain', 'Weltherrscher'))
>>> print db.to_str(keys, records)
::: Datensaetze :::
--------------------
ID: 0
Name: Erik
Beruf: Wiking
--------------------
ID: 1
Name: Brain
Beruf: Weltherrscher
--------------------
2 Datensaetze.
```

Hier könnte man noch die Schlüssel als Argument übergeben und überprüfen, ob die Länge von Schlüsseln und Werten übereinstimmt.

3.4.4 Index zu einer ID ermitteln

Da der Benutzer die IDs zum Zugriff auf Datensätze verwendet, braucht man für mehrere Operationen eine Möglichkeit von der ID auf den Index in der Liste mit den Datensätzen zu kommen.

```
def id2index(records, id_):
    for i, record in enumerate(records):
        if record[0] == id_:
            return i
    raise KeyError('record with id %s not found' % id_)
```

In der Testdatenbank sind die IDs identisch mit den Indexen:

```
>>> keys, records = TEST_DB
>>> db.id2index(records, 0)
0
>>> db.id2index(records, 1)
1
```

Das kann sich natürlich ändern wenn die Datenbank sortiert wird oder Datensätze gelöscht werden.

```
>>> db.id2index([TEST_ENTRY], 42)
0
```

Der Zugriff auf eine nicht existierende ID ist ein Fehler:

```
>>> db.id2index(records, 23)
Traceback (most recent call last):
    ...
KeyError: 'record with id 23 not found'
```

3.4.5 Datensatz per ID holen

Jetzt kann man auf einen Datensatz mittels ID zugreifen.

```
def get_record(records, id_):
    return records[id2index(records, id_)]
```

Anwendung:

```
>>> db.get_record(records, 1)
(1, ('Brain', 'Weltherrscher'))
>>> db.get_record(records, 23)
Traceback (most recent call last):
    ...
KeyError: 'record with id 23 not found'
```

3.4.6 Datensatz mit ID speichern

Wenn man einen vorhandenen Datensatz verändern und wieder zurück in die Datenbankstruktur schreiben möchte, muss sichergestellt sein, dass der Datensatz mit der alten ID überschrieben wird.

```
def put_record(records, record):
    id_ = record[0]
    records[id2index(records, id_)] = record
```

Eigentlich ist Brain noch nicht Weltherrscher, da hat er beim Ausfüllen des Formulars ein wenig übertrieben. Holen wir ihn mal auf den Boden der Tatsachen zurück:

```
>>> keys, records = TEST_DB
>>> db.put_record(records, (1, ('Brain', 'Laborratte')))
>>> print db.to_str(keys, records)
::: Datensaetze :::
--------------------
ID: 0
Name: Erik
Beruf: Wikinger
--------------------
ID: 1
Name: Brain
Beruf: Laborratte
--------------------
2 Datensaetze.
```

Datensätze mit einer ID, die nicht in der Datenbank existiert, lassen sich nicht auf diese Weise in die Datenbank eintragen:

```
>>> db.put_record(records, (23, ('Existiert', 'nicht')))
Traceback (most recent call last):
    ...
KeyError: 'record with id 23 not found'
```

Die Ausnahme bei einer nicht existenten ID ist vielleicht etwas übertrieben. Man könnte in dem Fall auch einfach den Datensatz hinzufügen.

3.4.7 Datensatz entfernen

Man möchte natürlich auch Datensätze über ihre ID löschen können.

```
def remove_record(records, id_):
    del records[id2index(records, id_)]
```

Falls Brain bei dem Versuch die Weltherrschaft zu erlangen, umkommen sollte:

```
>>> keys, records = TEST_DB
>>> # Kopie erstellen, damit die Testdatenbank nicht veraendert wird:
>>> records = list(records)
>>> db.remove_record(records, 1)
>>> print db.to_str(keys, records)
::: Datensaetze :::
--------------------
ID: 0
Name: Erik
Beruf: Wikinger
--------------------
1 Datensatz.
```

Etwas zu löschen, was es nicht gibt, schlägt natürlich fehl:

```
>>> db.remove_record(records, 23)
Traceback (most recent call last):
    ...
KeyError: 'record with id 23 not found'
```

3.4.8 Laden und Speichern

Für die Datenspeicherung ist die unkomplizierteste Methode das *pickle*-Modul, mit dem man nahezu beliebige Python-Objekte speichern und laden kann.

```
def save(keys, records, filename):
    db_file = open(filename, 'wb')
    pickle.dump((keys, records), db_file)
    db_file.close()

def load(filename):
    db_file = open(filename, 'rb')
    keys, records = pickle.load(db_file)
    db_file.close()
    return keys, records
```

Die Testdatenbank einmal speichern und danach wieder laden:

```
>>> keys, records = TEST_DB
>>> db.save(keys, records, 'test.dat')
>>> keys, records = db.load('test.dat')
>>> keys
('name', 'beruf')
>>> records
[(0, ('Erik', 'Wikinger')), (1, ('Brain', 'Laborratte'))]
```

3.4.9 Suchen in der Datenbank

Die Suche in der Datenbank stützt sich auf die Suche in einem einzelnen Datensatz.

```
def search(records, keyword):
    return [record for record in records if record_contains(record, keyword)]
```

Eine Suche ohne Ergebnis, eine nach Erik und eine deren Suchbegriff in allen beiden Datensätzen vorkommt:

```
>>> key, records = TEST_DB
>>> db.search(records, 'nix')
[]
>>> db.search(records, 'erik')
[(0, ('Erik', 'Wikinger'))]
>>> db.search(records, 'R')
[(0, ('Erik', 'Wikinger')), (1, ('Brain', 'Laborratte'))]
```

3.4.10 Sortieren der Datenbank

Die Sortierung geht nach den Werten in den Datensätzen. Es wird alphabetisch aufsteigend nach den einzelnen Feldern sortiert, d.h. das erste Feld ist das wichtigste. Sollten dort zwei gleiche Werte stehen, wird das nächste Feld als Vergleichskriterium benutzt. Das ist das Standardverhalten beim Vergleich von Tupeln.

```
def sort_db(records):
    records.sort(key=lambda (id_, values): values)
```

Erik und Brain tauschen ihre Plätze:

```
>>> key, records = TEST_DB
>>> db.sort_db(records)
>>> print db.to_str(keys, records)
::: Datensaetze :::
--------------------
ID: 1
Name: Brain
Beruf: Laborratte
--------------------
ID: 0
Name: Erik
Beruf: Wikinger
--------------------
2 Datensaetze.
```

3.5 Benutzerschnittstelle

Hier ist ein etwas härterer Schnitt als zwischen den beiden letzten Abschnitten. Während einzelne Datensätze und die Datenbank sehr eng verknüpft sind, ist die Benutzerschnittstelle etwas, das abgetrennt werden kann und sollte. Dazu wäre eigentlich ein eigenes Modul für das Hauptprogramm angemessen. Da sich hier Benutzerschnittstelle und Datenbankfunktionalität einen Namensraum teilen, wird zumindest einigen der Funktionen der Benutzerschnittstelle der Präfix *ui_* vorangestellt, um Namenskollisionen zu vermeiden.

Die Funktionen, die einzelne Menüpunkte aus dem Hauptprogramm implementieren, haben alle die gleiche Signatur -- sie bekommen die Datenbank als Liste mit zwei Elementen, Schlüssel und Liste mit Datensätzen, als Argument. Damit lassen sich die Menüpunkte als einfache Datenstruktur anlegen und somit einfach erweitern.

3.5.1 Werte bearbeiten

Wir brauchen eine Funktion, um einen Datensatz zu bearbeiten. Der alte Datensatz soll vorher ausgegeben werden und dann kann der Anwender zu jedem Schlüssel einen neuen Wert eingeben, oder den alten mit einer Leereingabe übernehmen. Dabei wird die Bedingung erzwungen, dass mindestens zwei Felder mit Werten belegt sein müssen. Zurückgegeben wird ein neues Tupel mit den neuen bzw. veränderten Werten.

```
def edit_values(keys, values):
    print '- Datensatz bearbeiten -'
    print 'Alter Datensatz:'
    print record2str(keys, (None, values))
```

```
    print
    while True:
        print 'Neueingabe (Leereingabe um alten Wert zu uebernehmen):'
        result = list()
        for key, old_value in zip(keys, values):
            new_value = raw_input('%s: ' % key.capitalize())
            result.append(new_value or old_value)
        if at_least_filled(result, 2):
            return tuple(result)
        else:
            print 'Fehler: Es müssen mindestens 2 Felder gefüllt werden.'
```

3.5.2 Sicherheitsabfrage

Eine Hilfsfunktion für Sicherheitsabfragen beim Speichern oder Löschen der gesamten Datenbank. Die Funktion liefert *True* falls die Antwort des Benutzers nicht mit einem grossen oder kleinen 'n' beginnt.

Damit es also wirklich eine *Sicherheitsabfrage* ist, muss die Frage so formuliert werden, dass *Ja* die „sichere" Antwort ist.

```
def ask_yes_no(prompt):
    return not raw_input(prompt + ' (J/n) ').lower().startswith('n')
```

Hier könnte man natürlich eine flexiblere Funktion schreiben, bei der man auch auswählen kann, was die bevorzugte Antwort ist.

3.5.3 Neu anlegen

Eine neue Datenbank anlegen. Sollten sich Werte im Speicher befinden, so werden diese nach Rückfrage vorher gesichert.

```
def new(database):
    keys, records = database
    if records and ask_yes_no('Alte Werte vorher speichern?'):
        ui_save(database)
    database[:] = (ADDRESS_KEYS, list())
```

3.5.4 Laden

Nach Abfrage des Dateinamens wird eine Datei geladen. Ein Fehler beim Laden wird dem Benutzer angezeigt.

```
def ui_load(database):
    try:
        database[:] = load(raw_input('Dateiname: '))
    except IOError, error:
        print 'Fehler beim Laden:'
        print error
```

3.5.5 Speichern

Die Daten werden in eine Datei gespeichert. Der Vorgang wird so oft wiederholt, bis dass Speichern fehlerfrei geklappt hat.

```
def ui_save((keys, records)):
    while True:
        try:
            save(keys, records, raw_input('Dateiname: '))
            break
        except IOError, error:
            print 'Fehler beim speichern:'
            print error
```

Hier könnte man eine Möglichkeit einbauen, die Endlosschleife auf Wunsch auch ohne erfolgreichen Speichervorgang zu verlassen. Zum Beispiel durch eine Leereingabe, da man Dateien mit so einem Namen auf den meisten Dateisystemen nicht anlegen kann.[2]

3.5.6 Anzeigen

Die Anzeige ist sehr einfach, da die meiste Arbeit schon von den Funktionen für die Datenbank bzw. einzelnen Datensätzen übernommen wird. Obwohl das strenggenommen nicht deren Aufgabe ist.

```
def display((keys, records)):
    print to_str(keys, records)
    raw_input('Eingabetaste zum Fortsetzen...')
```

In einer etwas komfortableren Oberfläche würde diese Funktion nicht nur die komplette Darstellung übernehmen, sondern auch die Datensätze „seitenweise“ am Bildschirm anzeigen, so dass man nicht den Scrollbalken des Terminals benutzen muss, wenn die Datenbank mehr Datensätze enthält als auf den Bildschirm passen.

3.5.7 Datensatz eintragen

Zum Eintragen eines neuen Datensatzes wird ein leerer Datensatz zum bearbeiten angezeigt und das Ergebnis dieser Bearbeitung dann in die Datenbank geschrieben.

```
def enter_new((keys, records)):
    add_values(records, edit_values(keys, ('',) * len(keys)))
```

3.5.8 Datensatz entfernen

Ein Datensatz kann anhand seiner ID aus der Datenbank entfernt werden.

```
def remove((keys, records)):
    try:
        remove_record(records, int(raw_input('ID: ')))
    except KeyError:
        print 'Unbekannte ID!'
```

[2] Beim Dateisystem der Laufwerke für den *Commodore C64* kann man so einen Dateinamen anlegen. :-)

3.5.9 Bearbeiten

Der Benutzer wird nach der ID des Datensatzes gefragt, den er bearbeiten möchte. Nachdem er bearbeitet wurde, werden die veränderten Werte zurückgeschrieben.

```
def edit((keys, records)):
    try:
        id_, values = get_record(records, int(raw_input('ID: ')))
        put_record(records, (id_, edit_values(keys, values)))
    except KeyError:
        print 'Unbekannte ID!'
```

3.5.10 Suchen

Der Benutzer wird nach einem Suchbegriff gefragt, nach dem die Daten dann durchsucht werden.

```
def ui_search((keys, records)):
    result = search(records, raw_input('Suchbegriff: '))
    if result:
        print to_str(keys, result)
    else:
        print 'Leider keine Treffer!'
```

3.5.11 Programmende

Vor Programmende wird noch einmal nachgefragt, ob die Daten gespeichert werden sollen.

```
def ui_exit(database):
    question = 'Soll die Datenbank vor dem Beenden gespeichert werden?'
    if ask_yes_no(question):
        ui_save(database)
    sys.exit()
```

3.5.12 Menü

Die Menüeinträge sind Tupel der Form *(Buchstabe, Text, Funktion)*, wobei nur die ersten beiden Elemente hier auch wirklich benötigt werden. Der Buchstabe und der Text zu jedem Eintrag werden ausgegeben und am Ende kommt eine Eingabeaufforderung, bei der noch einmal alle Buchstaben in einer Liste angegeben werden.

```
def menue(entries):
    keys = list()
    print '> Menue:'
    for key, text, dummy in entries:
        print '%s - %s' % (key, text)
        keys.append(key)
    return raw_input('Auswahl (%s): ' % ', '.join(keys))
```

3.5.13 Hauptprogramm

Im Hauptprogramm wird die Datenbank angelegt. Und eine Struktur *menue_entries*, die alle notwendigen Daten für die Menüpunkte enthält. Das sind: der Buchstabe unter dem man den Menüpunkt aufrufen kann, ein beschreibender Text und die Funktion, die den jeweiligen Menüpunkt implementiert. Aus der Struktur wird eine Abbildung von Buchstabe nach Funktion erzeugt.

Dann folgt eine Haupschleife, die Benutzereingaben entgegen nimmt und die gewünschten Menüpunkte aufruft.

```
def main():
    database = [(), []]
    new(database)

    menue_entries = (('N', 'Neu anlegen', new),
                     ('l', 'Laden', ui_load),
                     ('s', 'Speichern', ui_save),
                     ('a', 'Auflisten', display),
                     ('n', 'Neuen Datensatz anlegen', enter_new),
                     ('e', 'Datensatz entfernen', remove),
                     ('b', 'Datensatz bearbeiten', edit),
                     ('S', 'Suchen', ui_search),
                     ('x', 'Programmende', ui_exit))
    key2func = dict((key, func) for key, dummy, func in menue_entries)
    assert len(menue_entries) == len(key2func)

    print '--- Adressverwaltung ---\n'
    while True:
        choice = menue(menue_entries)
        try:
            func = key2func[choice]
        except KeyError:
            print 'Falsche Einfabe!'
            continue
        func(database)

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

4 Nachbemerkungen

4.1 Funktional oder imperativ

Das Programm sieht etwas nach funktionaler Programmierung aus, weil man ohne Klassen mit den typischen Datentypen auskommen muss, die auch bei funktionalen Programmiersprachen zur Verfügung stehen. Bei einer klassischen imperativen Sprache wie *C* oder *Pascal* hätte man für den Datensatz und die Datenbank jeweils ein `struct` bzw. `RECORD` deklariert. Also im Grunde eine Klasse ohne Methoden.

Es wäre vielleicht eine der einfachsten Verbesserungen so eine Klasse einzuführen. Das klassische *Bunch*-Rezept würde ausreichen und zumindest in der Anwendung wohl die Schüler auch nicht überfordern.

```
class Bunch(object):
    def __init__(self, **kwargs):
        self.__dict__ = kwargs

    def __repr__(self):
        kwargs = ', '.join('%s=%r' % item
                           for item in sorted(self.__dict__.iteritems()))
        return '%s(%s)' % (self.__class__.__name__, kwargs)
```

Ein einzelner Datensatz sieht dann so aus:

```
>>> record = db.Bunch(id=0, values=('Erik', 'Wikinger'))
>>> record.id
0
>>> record.values
('Erik', 'Wikinger')
```

Und die Datenbank so:

```
>>> database = db.Bunch(keys=('name', 'beruf'), records=[record])
>>> database.keys
('name', 'beruf')
>>> database.records
[Bunch(id=0, values=('Erik', 'Wikinger'))]
```

Es wäre auch möglich jedem Datensatz eine Referenz auf die Schlüssel der Datenbank mitzugeben, dann bräuchte man die Schlüssel nicht separat an Funktionen, die auf einzelnen Datensätzen operieren, übergeben.

4.2 Laufzeit

Wegen des Zugriffs über IDs haben viele Operationen eine lineare Laufzeit. Weil die Daten auch sortiert werden, kommt ein Dictionary mit einer Abbildung von IDs auf Datensätze aber nicht in Frage. Man müsste also beide Datenstrukturen parallel verwalten, um schnellen Zugriff über die ID *und* eine geordnete Liste von Datensätzen zu haben. Das ist bei der imperativen Variante des Programms allerdings etwas unschön, da die Kapselung hier noch informeller wäre, als das bei Klassen in Python schon der Fall ist.

Bei der zu erwartenden Grösse der Datenbanken im vorliegenden Programm, ist das aber nicht ganz so tragisch.

4.3 Erweiterungen

Das Hauptprogramm benutzt als Beispiel nur Adressen. Hier könnte man den Programmpunkt zum Anlegen einer Datenbank um die Eingabe von beliebigen Feldern erweitern.

Bei der Suche könnte man das Einschränken auf bestimmte Felder oder eine Suche mittels regulärer Ausdrücke hinzufügen.

5 Über das Dokument

Dieses Dokument ist in reStructuredText gesetzt und enthält neben dem beschreibenden Text den kompletten Quelltext der Minidatenbank. Ein lauffähiges Python-Modul kann man mittels pylit erstellt werden und die angegebenen Beispiele am Python-Prompt lassen sich mit dem *doctest*-Modul überprüfen.