- **Du möchtest numerische Daten speichern (Integer oder Fließkommazahlen), wobei eine effiziente Speichernutzung und die Leistung sehr wichtig sind?** Probiere aus, ob dir `array.array` alles bietet, was du brauchst. In einem solchen Fall kannst du dich auch außerhalb der Standardbibliothek umsehen und Pakete wie `NumPy` oder `Pandas` verwenden.
- **Du hast Textdaten in Form von Unicode-Zeichen?** Verwende den integrierten Python-Datentyp `str`. Wenn du einen veränderbaren String brauchst, nimm eine Liste aus Zeichen.
- **Du willst einen zusammenhängenden Block von Bytes speichern?** Verwende den unveränderbaren Typ `bytes`. Wenn du eine veränderbare Datenstruktur benötigst, nimm `bytearray`.

In den meisten Fällen nehme ich zu Anfang eine einfache Liste und wechsle später zu einem spezielleren Typ, falls die Leistung oder der Speicherplatz eine Rolle spielen. Meistens bietet eine Allzweck-Arraydatenstruktur wie `list` die höchste Entwicklungsgeschwindigkeit und auch den größten Komfort zum Programmieren. Meiner Erfahrung nach ist das zu Anfang viel wichtiger, als gleich zu Beginn zu versuchen, auch noch das letzte Quäntchen Leistung herauszuholen.

5.3 Datensätze, Strukturen und DTOs

Datensätze enthalten eine feste Anzahl von Feldern, die jeweils einen eigenen Namen haben und auch einen eigenen Typ aufweisen können. In diesem Abschnitt sehen wir uns an, wie du in Python Datensätze, Strukturen und einfache, klassische Datenobjekte mit den integrierten Typen und den Klassen aus der Standardbibliothek implementierst. Übrigens verwende ich den Begriff »Datensatz« (*record*) hier in einer sehr weit gefassten Bedeutung. Beispielsweise bespreche ich hier auch Typen wie Tupel, die nicht unbedingt Datensätze im strengen Sinne sind, da sie keine benannten Felder enthalten. Python stellt verschiedene Datentypen zur Implementierung von Datensätzen, Strukturen und Datenübertragungsobjekten (Data Transfer Objects, DTOs) bereit. In diesem Abschnitt sehen wir uns die einzelnen Implementierungen und ihre jeweiligen Eigenschaften an. Am Ende erhältst du eine Zusammenfassung und einen Leitfaden für die Auswahl.

Fangen wir an!

Einfache Datenobjekte: dict

Python-Dictionarys können eine beliebige Anzahl von Objekten enthalten, die jeweils durch einen eindeutigen Schlüssel identifiziert werden.[15] Sie werden auch als *Maps* und *assoziative Arrays* bezeichnet und ermöglichen das effiziente Nachschlagen, Einfügen und Löschen beliebiger Objekte, die mit einem Schlüssel verknüpft sind. Es ist möglich, Dictionarys als Datentyp oder Datenobjekt für Datensätze zu verwenden. Sie lassen sich leicht erstellen, da eine gewisse syntaktische

15 Siehe Abschnitt »Dictionarys«

```
    mileage: float
    automatic: bool

>>> car1 = Car('red', 3812.4, True)

# Instanzen haben eine übersichtliche Darstellung:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

# Zugriff auf Felder:
>>> car1.mileage
3812.4

# Felder sind unveränderbar:
>>> car1.mileage = 12
AttributeError: "can't set attribute"
>>> car1.windshield = 'broken'
AttributeError:
"'Car' object has no attribute 'windshield'"

# Typanmerkungen werden ohne ein zusätzliches Typüberprüfungswerkzeug
# wie mypy nicht durchgesetzt:
>>> Car('red', 'NOT_A_FLOAT', 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

Serialisierte C-Strukturen: struct.Struct

Die Klasse `struct.Struct` wandelt Python-Werte in serialisierte C-Strukturen aus `bytes`-Objekten um.[23] Damit kannst du beispielsweise Binärdaten handhaben, die in Dateien gespeichert sind oder über Netzwerkverbindungen hereinkommen. Solche Strukturen werden mit einer Minisprache ähnlich wie für Formatstrings definiert, mit der du die Anordnung verschiedener C-Datentypen wie `char`, `int` und `long` sowie ihrer vorzeichenlosen Varianten (`unsigned`) festlegen kannst. Serialisierte Strukturen werden nur selten für Datenobjekte verwendet, die ausschließlich im Python-Code gehandhabt werden sollen. Sie dienen hauptsächlich als Datenaustauschformat und nicht als eine Möglichkeit, um Daten, die nur im Python-Code verwendet werden, im Arbeitsspeicher festzuhalten.

Wenn du primitive Daten in Strukturen verpackst, kannst du damit manchmal weniger Arbeitsspeicher verbrauchen als bei der Verwendung anderer Datentypen. Meistens ist eine solche Optimierung jedoch ziemlich anspruchsvoll und wahrscheinlich unnötig.

```
>>> from struct import Struct
>>> MyStruct = Struct('i?f')
>>> data = MyStruct.pack(23, False, 42.0)
```

23 Siehe Python-Dokumentation, »struct.Struct«