Python richtig lernen

Christian Kaiser und Stefan Keller

14.-15. Januar 2015

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Automatisierung von Prozessen und Abläufen hilft die Effizienz zu steigern und den Arbeitsablauf der Datenverarbeitung zu vereinfachen. Zudem können mit relativ einfachen Mitteln neue bis anhin nur schwer realisierbare Verarbeitungsschritte umgesetzt werden. Python ist eine leistungsfähige, erweiterbare und doch einfach zu lernende Programmiersprache für ein breites Anwendungsspektrum. Mit Python lassen sich komplette Desktop-Programme oder umfangreiche Web-Applikationen entwickeln. Aber Python eignet sich auch sehr gut als einfache Skriptsprache und wird in vielen Programmen zu diesem Zweck eingesetzt. Beispiele sind OpenOffice, Blender oder GIMP (Grafikprogramme), SPSS (Statistiksoftware), QGIS oder ArcGIS (Geographische Informationssysteme), und viele andere.

Dieser Kurs bietet eine Einführung in Python vorwiegend mit dem Ziel kleine Programme oder Skripts zu schreiben, gibt aber auch die nötigen Grundlagen um an grösseren Python-Projekten zu arbeiten.

1.1 Lernziele

Nach Abschluss des Kurses sind Sie in der Lage, eigenständig einfache Python-Skripts zu schreiben, um spezifische Arbeitsabläufe der Datenverarbeitung zu automatisieren.

1.2 Zielgruppe

Der Kurs richtet sich an Personen, die Python lernen möchten, um einfach und effizient kleine Skripts zu programmieren. Der Kurs wird mit Beispielen aus verschiedenen Anwendungsgebieten illustriert. Der Kurs setzt keinerlei Kenntnisse einer Programmiersprache voraus. Ein gewisses Abstraktionsvermögen und der Wille, in die Informatikwelt einzutauchen, sind jedoch von Vorteil. Vom Kurs können auch Personen profitieren, die Python bereits kennen, da Sie unter moderner Softwareentwicklung u.a. auch objektorientiertes Programmieren und UNIT-Testing kennenlernen.

1.3 Was ist eine Skript-Sprache?

Eine Skript-Sprache ist eine relativ einfach zu benützende Programmiersprache. Skripts sind kleine Programme die es erlauben eine relativ einfache Aufgabe automatisch durchzuführen.

Programmiersprachen wie C oder C++ erlauben es Code auf einem sehr detaillierten Niveau zu schreiben, der dann kompiliert (in Maschinensprache übersetzt) und ausgeführt werden kann. Solche Programme laufen in der Regel sehr schnell. Meistens ist es jedoch nicht notwendig, extrem schnelle Programme zu haben, die aufwändig programmiert werden müssen. Es ist in vielen Fällen angepasster eine einfachere Programmiersprache zu verwenden, die schnelles Programmieren erlaubt, jedoch in der Ausführung etwas langsamer ist. Dafür wurden Skriptsprachen konzipiert. Es ist jedoch nicht so, dass Skriptsprachen ausschliesslich für kleine Programme verwendet werden können. Skriptsprachen sind heutzutage sehr ausgereift und Computer in der Regel sehr leistungsstark, was den Gebrauch von solchen Sprachen auch für grössere Projekte erlaubt.

Skriptsprachen laufen in der Regel in einem sogenannten Interpreter, der die Instruktionen während der Programmausführung in Maschinensprache übersetzt (in Realität gibt es viele Mischformen zwischen reinem Interpreter und kompilierter Code, zum Beispiel Java oder Python, aber wir müssen uns darum gar nicht kümmern). Ein Interpreter erlaubt auch das einfache 'Experimentieren', also Ausführung Linie um Linie, wie in einem Terminal-Computer. Wir werden Python ebenfalls durch solches Experimentieren lernen.

1.4 Warum Python?

Die wahrscheinlich erste populäre Skriptsprache war Perl, das auch heute noch weit verbreitet ist. Doch Python hat gewisse Vorteile gegenüber Perl oder anderen Programmiersprachen wie C++ oder Java. Erstens ist Python eine sehr klare und elegante Sprache. Python-Programme sind meist besser lesbar als Perl- oder Java-Programme, und es werden häufig weniger Zeilen Code benötigt um die gleiche Aufgabe zu bewältigen (das heisst weniger Möglichkeiten für Fehler!). Zudem läuft Python auf allen gängigen Betriebssystemen. Python ist erweiterbar durch sogenannte Module, die in Python selbst geschrieben sind, oder in einer Programmiersprache wie C oder C++. Dadurch können Programmteile die extrem schnell laufen müssen auch in C geschrieben werden. Diese Erweiterbarkeit von Python erlaubt es auch, Python als Skriptsprache für traditionelle in C oder C++ geschriebene Programme zu verwenden.

1.5 Lernumgebung

Um den Lernprozess so einfach wie möglich zu gestalten, werden wir in diesem Kurs eine virtuellen Computer einsetzen, auf dem alle nötigen Komponenten für Python schon vorinstalliert sind. Dies erlaubt es uns, uns auf das Lernen der Programmiersprache zu kümmern ohne zuerst den Installationsprozess zu durchlaufen. Installationen können unter Umständen frustrierend sein, speziell für Anfänger.

Ein virtueller Computer ist eine Art zweiter Computer der innerhalb eines speziellen Programms läuft. Dieser virtuelle Computer hat alles was ein richtiger Computer auch hat: eine Festplatte, Arbeitsspeicher, einen Monitor (ein Fenster), ein Betriebssystem, etc. Alles was wir machen müssen um einen solchen virtuellen Computer auf unserem richtigen Computer zu installieren, ist das entsprechende Programm zu installieren, und Arbeitsspeicher und Festplatte (eine ziemlich grosse Datei) zu definieren. Das Programm das wir für diese 'Virtualisierung' brauchen ist *VirtualBox* von Oracle (`www.virtualbox.org`).

Um Python effektiv zu erlernen, sollten wir folgende Punkte nicht vergessen:

- Ausprobieren kostet nichts! Um zu wissen ob etwas funktioniert oder nicht gibt es nichts einfacheres als ausprobieren. Am Besten probie-

ren wir gleich verschiedene Varianten bis es eben funktioniert. Hier ist Kreativität gefragt!

- Sie können Ihren Computer nicht kaputt machen mit programmieren. Im schlimmsten Fall gibt es einen Fehler, und das ist wirklich nicht schlimm! Es ist übrigens sogar so, dass selbst geübte Programmierer fast mehr Fehler produzieren als Erfolge...

- Nicht vergessen: GIYF (Google Is Your Friend). Im Falle eines Problems, sind Sie meist nicht der erste. Häufig finden Sie schnell eine Antwort im Web. Falls Sie die englische Sprache beherrschen, ist häufig `www.stackoverflow.com` eine gute Adresse um Antworten auf Ihre Fragen zu finden. Im Allgemeinen gilt auch hier: versuchen Sie verschiedene Varianten für Ihre Suchanfrage. Also auch hier ist etwas Kreativität gefragt.

1.6 Python starten

Python kann auf verschiedene Weisen genutzt werden. Dabei gibt es Unterschiede zwischen den Betriebssystemen und der Art der Installation. Um mit Python anzufangen geben wir nur an, wie wir Python unter Linux starten und erste Schritte damit unternehmen. Zuerst müssen wir das Terminal-Programm öffnen. Das befindet sich üblicherweise im 'Applications'-Menü. Anschliessend schreiben wir einfach *'python'* in Terminal und drücken 'Enter'. So wird die Python-Konsole gestartet, die anschliessend auf unsere interaktiven Befehle wartet.

In Linux gibt es noch eine etwas funktionsreichere Python-Konsole die sich besser für interaktive Befehle eignet. Es handelt sich um iPython (für 'interactive Python'). Anstatt *'python'* einfach *'ipython'* im Terminal eingeben. Wir werden für die ersten Schritte die iPython-Konsole benützen.

Um die Python-Konsole zu verlassen, müssen wir die Tastenkombination Ctrl-D benützen.

Etwas später werden wir auch schauen, wie man Python-Code der in Dateien gespeichert ist als kleines Programm ausführen kann.

Kapitel 2

Grundlagen in Python

2.1 Ein einführendes Beispiel

Fangen wir gleich mit konkreten Beispielen an. Im Python-Interpreter, geben wir Linie um Linie ein:

```
1 print "Hello world"
2 print 'Hello', "world"
3 print 5 + 10
4 print "5 + 10 =", 5 + 10
5
6 # Eine kleine Schlaufe:
7 for i in range(10):
8     print i*i
```

Dieses kleine Beispiel enthält schon mehrere wichtige Konzepte:

- Als erstes haben wir das Schlüsselwort **print**. In Python gibt es ein paar wenige Schlüsselwörter, die meisten davon werden wir noch sehen. **print** gibt einfach den nachfolgenden Inhalt in der Konsole aus.
- Zeichenketten (Strings) werden durch Anführungszeichen oder Apostrophe abgegrenzt. In Python sind beide identisch.
- Mit **print** können mehrere Elemente gleichzeitig ausgegeben werden, indem sie einfach durch ein Komma abgetrennt werden.
- Es können einfach Berechnungen durchgeführt werden (Python als Taschenrechner...).

- Ein einzelner Python-Befehl geht bis zum Zeilenende. In Python gibt es keine Zeichen die einen Befehl beenden, wie zum Beispiel ';' in C oder Java.

- Kommentare beginnen mit '#' und gehen bis zum Zeilenende. Und wie in allen Programmiersprachen gilt auch hier: Kommentare helfen das Programm zu verstehen, also schreiben Sie so viele Kommentare wie möglich damit Sie (und Ihre Grossmutter!) auch in 2 Jahren verstehen um was es geht!

- Schlaufen repetieren Code. In Python geht das mit dem Schlüsselwort **for ... in ...:**. In unserem Beispiel ist **range(10)** eine 'Liste' mit Werten von 0 bis 9 (10 minus 1), und die Variable **i** nimmt jeden dieser Wert nacheinander an und führt jedes Mal den Befehl **print i*i** in Linie 8 aus.

- Die Linie 8 ist ein 'Code-Block'; dies ist der Code der sich innerhalb der Schlaufe befindet. In unserem Beispiel, der Code-Block besteht aus bloss einem Befehl. In Python werden Code-Blocks durch Einrückung definiert (vorangestellte 'Leerzeichen', also ein oder mehrere Tabulatoren oder Leerschläge; typischerweise jedoch genau 4 Leerschläge). Also: **in Python sind Leerschläge wichtig!** Und: mixen Sie nicht Tabulatoren und Leerschlägen, sonst ist Chaos vorprogrammiert. Das Einrücken von Code-Block ist eine der Charakteristiken von Python, das selten in anderen Programmiersprachen zu finden ist. Jedoch entsteht dabei ein Code, der generell besser lesbar ist. Es finden sich auch je länger je mehr Programmiersprachen die diese Charakteristik übernehmen (z.B. CoffeeScript).

2.2 Achte auf Details!

Beim Programmieren ist es extrem wichtig auf Details zu achten. Computer kennen keine Gnade bei Flüchtigkeitsfehlern oder Unachtsamkeit. Ein Tabulator ist nicht gleich 4 Leerschlägen, auch wenn es gleich aussieht. Ein fehlender Doppelpunkt oder ein Strichpunkt anstatt einem Komma ist einfach falsch und Python wird es Ihnen zu verstehen geben. Allerdings wird Python auch versuchen, den Fehler so gut wie möglich zu identifizieren und es Ihnen mitteilen.

Generell ist es wichtig, dass Sie den Code nicht einfach mit Kopieren-Einsetzen schreiben, sondern ihn selbst tippen. Damit werden Sie auf kleine Details aufmerksam die Ihnen sonst entgehen würden.

2.3 Variablen und Typen

Variablen können irgendwelche Werte enthalten, die manipuliert und an anderer Stelle gebraucht werden können. In Python werden Variablen ganz einfach definiert:

```
1 v = 9.5
2 zahl = 45
3 andere_zahl = 123
4 NochEineAndereZahl = v + zahl + 78
```

Variablen haben einen Namen, der aus einem einfachen Buchstaben, oder einer Kombination von Buchstaben, Zahlen und ein paar anderen erlaubten Zeichen bestehen kann. Ein viel benutztes Zeichen ist der 'Underscore' (_). Punkt, Komma oder Leerzeichen ist jedoch nicht erlaubt. Grundsätzlich können für Variablennamen Gross- und Kleinbuchstaben verwendet werden. Eine Konvention in Python sagt jedoch, für Variablennamen **nur Kleinbuchstaben** zu brauchen. Ein Variablennamen kann nicht mit einer Zahl beginnen. Auch sind Variablen die mit zwei 'Underscores' beginnen per Konvention für Python's internen Gebrauch reserviert.

Sobald eine Variable einen Wert bekommt, definiert Python auch einen **Typ** für die Variable. Der Variablen-Typ hat einen entscheidenden Einfluss welche Operationen gestattet sind mit dem Wert der Variablen, und was das Resultat ist. So gibt zum Beispiel die Operation 'addieren' auf Zahlen angewandt die Summe (3 + 4 = 7), aber auf Zeichenketten angewandt die Kombination dieser Zeichenketten ('abc' + 'def' = 'abcdef'). Dementsprechend ergibt eine Kombination von Zeichenketten und Zahlen bei der 'Plus-Operation' keinen Sinn (also 4 + '5' gibt einen Fehler, da '5' als Zeichenkette behandelt wird).

Gewisse Programmiersprachen wie zum Beispiel Javascript würden bei inkompatiblen Variablen-Typen eine Konversion in einen anderen Typ versuchen (so genannt 'weekly typed'). Python macht dies jedoch ganz bewusst nicht um Fehler zu vermeiden (so genannt 'strongly typed').

Es gibt ganz verschiedene dieser Variablen-Typen in Python, hier sind ein paar Wichtige:

- **String**: Eine Zeichenkette. Eine Zeichenkette beginnt und endet mit einem Anführungszeichen ("...") oder Apostroph ('). Solche Zeichenketten sind auf eine Zeile begrenzt. Um Zeichenketten über mehrere Zeilen zu definieren können 3 Anführungszeichen oder Apostrophe verwendet werden:

```
1 # Ein String mit mehrere Zeilen:
2 ein_text = """Diese Zeichenkette geht
3     ueber mehrere Zeilen"""
```

- **Integer**: Ein 'Integer' ist eine ganze, positive oder negative Zahl. Also zum Beispiel 4, 5, 0, -99 sind Integer-Zahlen. Mit Integer-Zahlen können alle gängigen Rechenoperationen durchgeführt werden. Das Resultat einer Rechenoperation mit Integer-Zahlen ist immer auch eine Integer-Zahl. Diese Tatsache ist eigentlich logisch, führt jedoch immer wieder zu Fehlern:

```
1 45 / 10      # Resultat: 4 !!!
```

  Als Resultat würde man hier selbstverständlich 4.5 erwarten, und nicht 4. Aber 4.5 ist keine ganze Zahl, also wird einfach der Dezimalteil abgeschnitten!

- **Floating-point number**: Eine 'Floating-point number', oder 'Fliesskommazahl' ist eine Zahl die Dezimalstellen erlaubt. Dabei ist es aber nicht so, dass jede beliebige Dezimalzahl dargestellt werden kann. Dies liegt an der Art wie diese Zahlen vom Computer gespeichert werden. Es kann dabei zu extrem kleinen Rundungsfehlern führen, welche in der Praxis meist jedoch nicht relevant sind.

  Floating-point numbers werden häufig einfach als 'Floats' bezeichnet. Mit Floats können natürlich auch alle gängigen Rechenoperationen durchgeführt werden, und das Resultat ist immer ein Float. Werden Integer und Floats kombiniert, ist das Resultat immer ein Float. So kann man einfach schreiben:

```
45 / 10.0   # Resultat: 4.5
```

Somit ist das Resultat nun korrekterweise 4.5. Dabei ist der Dezimalpunkt wichtig, die 0 nach dem Punkt kann man auch weglassen:

```
45 / 10.        # Resultat ist auch 4.5
```

- **Liste**: Eine Liste ist eine geordnete Sammlung von mehreren Variablen. Dabei können die enthaltenen Variablen verschiedene Typen haben. Eine Liste kann auch eine andere Liste enthalten. Eine Liste beginnt und endet mit einer eckigen Klammer, die verschiedenen Elemente werden mit Kommas getrennt:

```
[1, 2, 3, 4]              # Eine Liste mit Integer
[5, 4.5, "Hallo!"]        # Eine gemischte Liste
[4, 5, [1, 3.14, "abc"]]  # Eine Liste in einer Liste
```

Eine **for**-Schlaufe erlaubt es, für jedes Element einer Liste einen Code-Block auszuführen. Die Funktion **range()** erlaubt es einfach eine Liste mit einer Serie von Integer-Zahlen zu generieren:

```
range(5)
    # Gibt eine Liste mit 5 Elementen: [0,1,2,3,4]
```

- **Tupel**: Ein 'Tupel' ist so ziemlich das Gleiche wie eine Liste, ausser dass bei einem Tupel keine Elemente hinzugefügt oder entfernt werden können wenn es einmal erstellt ist, und die Elemente nicht verschoben werden können. Somit kann ein Tupel zum Beispiel nicht sortiert werden. Erstellt wird ein Tupel mit runden Klammern anstatt eckigen:

```
(1, 2, 3, 4)        # Ein Tupel mit Integer
(5, 4.5, "Hallo")   # Ein gemischtes Tupel
(4, 5, (1, 2.5))    # Ein Tupel in einem Tupel
(4, 5, [5, 6])      # Eine Liste in einem Tupel
```

Es ist noch anzufügen, dass die runden Klammern in vielen Fällen optional sind:

```
1     1, 2, 3, 4          # Das ist auch ein Tupel...
```

Kehren wir nun kurz zu unserem einführenden Beispiel in 2.1 zurück. Dort haben wir zum Beispiel auf Linie 2 geschrieben:

```
1 print 'Hello', "world"
```

Das war nicht anderes als die Ausgabe mit **print** von einem Tupel bestehend aus den Elementen **'Hello'** und **'world'**!

- **Dictionary**: Ein Dictionnaire ist auch eine Sammlung von Elementen ähnlich wie eine Liste oder ein Tupel. Der Unterschied ist, dass der Dictionnaire jedem Element einen Namen zuordnet. Man spricht dabei von 'Key-Value'-Paaren. Der 'Key' ist der Schlüssel oder Name eines Elementes. Das sieht dann so aus:

```
1 {'name 1': 'wert 1', 'name 2': 'wert 2'}
2 {'Name': 'Hildegard', 'Alter': 22, 'Wohnort': 'Mathon'}
3 {1: 'abc', 'abc': 2, (1,3,4): [4, 5, 6]}
```

Ein Dictionnaire wird also mit den geschweiften Klammern definiert, einzelne Key-Value-Paare werden mit Kommas getrennt, und Key und Value werden durch einen Doppelpunkt getrennt (zuerst der Schlüssel und dann der Wert). Als Namen können alle nicht-änderbaren Variablentypen dienen; dazu gehören Strings, alle Zahlen (Integer und Floats) und Tupels (jedoch weder Liste noch Dictionnaire, da diese geändert werden können). Werte können irgend einen Variablentyp haben.

Einen Wert in einem Dictionnaire abfragen, ändern oder hinzufügen ist einfach:

```
1 d = {'a': 1, 'b': 2}  # d ist ein Dictionnaire
2 print d['a']          # Gibt 1 aus
3 d['d'] = 4            # Fuegt ein neues Paar hinzu
4 d['b'] = 22           # Aendert ein Wert
```

Es ist auch möglich eine **for**-Schlaufe zu machen mit einem Dictionnaire, der jedes Element durchläuft:

```
d = {'a': 1, 'b': 2, 'c': 3}
for key in d:
    # Key-Value-Paare eins um eins ausgeben:
    print key, '->', d[key]
```

Eine **for**-Schlaufe mit einem Dictionnaire gibt also jeden Key eins um eins aus; wir müssen den dazugehörigen Wert dann separat anfragen.

2.3.1 Konvertieren von Variablentypen

Da Python keine automatischen Typ-Konvertierungen vornimmt, müssen wir das selbst anhand von einigen speziellen Funktionen machen, sofern eine solche Konvertierung Sinn macht. Hier sind die wichtigsten Konvertier-Funktionen:

- **str** konvertiert einen Wert in einen String:

```
a = 4           # a ist ein Integer
b = str(a)      # b ist ein String
str([1,3,4])    # Liste als String...
```

- **int** und **float** konvertieren einen Wert in einen Integer bzw. Float, sofern möglich. Falls es nicht möglich ist, ergibt das einen Fehler (will heissen: also ausprobieren was möglich ist und was nicht!).

```
a = '4'      # a ist ein String
int(a)       # Gibt einen Integer mit Wert 4
float(a)     # Gibt einen Float mit Wert 4.0
int(['1', '2', '3'])    # FEHLER...
```

Die Konvertierung einer Liste in einen Integer geht nicht. Python versucht also nicht jedes Element in einer Liste einzeln zu konvertieren, sondern versucht es als Ganzes, was natürlich zum Scheitern verurteilt ist. Aber alle Werte in einer Liste einzeln zu konvertieren ist nicht schwierig, hier von String zu Integer:

```
map(int, ['1', '2', '3'])
```

Die Funktion **map** geht also durch jedes einzelne Element in der Liste und konvertiert jeden einzelnen Wert mit der Funktion **int**.

- **list** und **tuple** erlauben das Konvertieren von Listen und Tupels in beide Richtungen:

```
eine_liste = [1,2,3]
ein_tupel = tupel(eine_liste)
wieder_eine_liste = list(ein_tupel)
```

Es ist noch hinzuzufügen, dass die Konvertierung von einem Typen in sich selbst immer auch möglich ist, also zum Beispiel:

```
str('hallo!')
int(4)
float(5.3)
```

Dies wird häufig angewandt, wenn man nicht ganz sicher ist ob eine Variablen einen gewissen Typen hat oder nicht.

2.4 Python-Programme ausführen

Bis anhin haben wir alle Python-Befehle interaktiv in der Python-Konsole eingegeben. Dies ist gut um Python zu lernen oder um Befehle auszuprobieren. Aber es ist unpraktisch um ein längeres Programm auszuführen, oder es weiter zu geben. Um dies zu ermöglichen speichern wir unsere Python-Befehl ganz einfach in einer normalen Text-Datei, der wir einfach die Endung **.py** anstatt **.txt** geben. Diese Datei ist dann schon ein Python-Skript!

Um das Python-Skript auszuführen, müssen wir auch das Terminal öffnen. Dann schreiben wir einfach

```
python /pfad/zur/python/datei.py
```

2.4.1 Try it out!

Um das schnell auszuprobieren, öffnen wir den einfachen Text-Editor von Ubuntu (oder jeden anderen Text-Editor der diesen Namen verdient), und schreiben folgende Zeile Code:

```
1 print 'Hello world!'
```

Wir speichern diese Datei im Home-Ordner des Users unter dem Namen *'world.py'*. Dann öffnen wir das Terminal und schreiben *'python world.py'*.

2.5 Code it!

Schreiben Sie ein kleines Programm in einer Python-Datei das die ersten Hundert Quadrate berechnet und ausgibt als *'1 im Quadrat ist 1'*, *'2 im Quadrat ist 4'* usw.

2.6 Zeichenketten manipulieren

Eine häufige Anwendung einer Programmiersprache ist die Manipulation von Zeichenketten. Während Stringmanipulationen in Programmiersprachen wie C mühsam sind, geht das in Python in den meisten Fällen ganz einfach. Schauen wir uns doch die häufigsten Fälle mal etwas genauer an.

Einen Teil einer Zeichenkette zu extrahieren geht in Python ganz einfach:

```
1 a = "Eine Zeichenkette"
2 a[0]        # Das erste Zeichen (1. Zeichen = Index 0)
3 a[-1]       # Das letzte Zeichen
4 a[0:4]      # Die ersten 4 Zeichen
5 a[4:8]      # Die Zeichen 5 bis 8
6 a[2:]       # Zeichen 3 bis zum Ende
7 a[-5:]      # Die letzten 5 Zeichen
8 a[2:-5]     # Vom Zeichen 3 bis zum sechst-letzten Zeichen
9 a[5:4]      # Eine leere Zeichenkette...
```

Häufig müssen wir auch wissen ob ein gewisses Wort in einer Zeichenkette vorkommt. Auch das ist einfach:

```
"Zeichen" in "Zeichenkette"     # Ist True
```

Um zu wissen an welcher Stelle ein gewisses Wort gefunden wurde:

```
a.find('kette')     # Gibt 12
a.find('Ente')      # Gibt -1 da nicht gefunden
```

Und suchen/ersetzen ist auch nicht kompliziert. Es ist allerdings zu beachten, dass im folgenden Beispiel die Variable a unverändert bleibt, und die neue Zeichenkette in der Variable b gespeichert wird. Also haben wir nun eine Zeichenkette und eine Bergkette:

```
b = a.replace('Zeichen', 'Berg')
```

Um die Länge einer Zeichenkette zu erfahren:

```
len(a)      # Gibt 17
```

Wie schon gesehen können in Python ganz einfach Strings auch über mehrere Linien definiert werden:

```
a = """Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Cras et ipsum quis mi semper accumsan.
Integer pretium dui id massa.
Suspendisse in nisl sit amet urna rutrum imperdiet.
Nulla eu tellus."""
```

Eine Zeichenkette kann aufgetrennt werden in mehrere kleine Zeichenketten die dann in einer Liste gespeichert werden. Dabei kann ein Trenn-Zeichen angegeben werden wo die Zeichenkette aufgetrennt werden soll. Ein Beispiel sollte das klar machen:

```
# Trennen wir die Auflistung in die einzelnen Elemente:
autos = 'Ford, Audi, Alfa Romeo, Peugeot, BMW, Opel'
marken = autos.split(',')
```

Dieses Auftrennen geht auch problemlos mit mehreren Zeichen.

Umgekehrt geht es auch. Mehrere Zeichenketten in einer Liste können zu einer grossen Zeichenkette zusammengefügt werden. Setzen wir also die Autos wieder zusammen:

```
autos_neu = ','.join(marken)
```

Die Syntax ist etwas gewöhnungsbedürftig, aber effizient.

In einem String können wir auch gewisse Zeichen am Anfang und Ende löschen. Alle 'Whitespaces' (Leerschläge, Tabulatoren, Neue-Linie-Zeichen: n und
r) am Anfang und Ende löschen geht ganz einfach:

```
    '  Eine nicht so saubere Zeichenkette.    \t\n\r'.strip()
```

Wir können auch ein anderes Zeichen, oder mehrere löschen:

```
    'zeichenkette'.strip('ez')  # gibt 'ichenkett'
```

In Python stehen auch Funktionen zur Verfügung um die Gross- und Kleinschreibung in einem String zu ändern. Die Funktionen *upper*, *lower*, *capitalize* und *swapcase* können für diese Manipulationen verwendet werden. Beispiel:

```
    'pIz BUIN'.capitalize() # Gibt 'Piz buin'
```

2.6.1 Exkurs in fortgeschrittene Funktionen

Um alle Wörter in einem String gross zu schreiben muss man folgende scheinbar komplizierte Operation durchführen: String zuerst nach Wörter auftrennen (also eine Liste mit Wörtern produzieren, die Funktion *split* verwenden), dann jedes einzelne Wort gross schreiben (mit *capitalize*), und schliesslich die Wörter in der Liste wieder zusammenfügen (mit *join*). Es gibt dabei einen Stolperstein: wie können wir die Funktion *capitalize* auf alle Wörter in der Liste gleichzeitig anwenden? Dafür gibt es die Funktion *map*. Schritt für Schritt:

```
berge = 'pIz BUIN, piz mOrTeRaTsCh, piz BERNINA'
# Auftrennen der Woerter in eine Liste:
berge_liste = berge.split()
# Magie: Anwenden von capitalize auf jedes Wort der Liste:
berge_liste_2 = map(str.capitalize, berge_liste)
# Und wieder zusammensetzen:
berge2 = ' '.join(berge_liste_2)
```

Die Funktion *map* wendet dabei die Funktion *capitalize* die auf Strings (*str*) angewendet werden kann, auf jedes Element von *berge_liste* an und gibt die veränderte Liste zurück. Praktisch!

Häufig werden Funktionen ineinander verschachtelt geschrieben, um den Code wesentlich kompakter zu schreiben. Obiges Beispiel kann so umgeschrieben werden:

```
berge = 'pIz BUIN, piz mOrTeRaTsCh, piz BERNINA'
berge2 = ' '.join(map(str.capitalize, berge.split()))
```

2.6.2 Unicode!

Unicode ist ein internationaler Standard der es erlaubt Buchstaben aus allen Alphabeten (lateinisch auch mit Umlauten und Akzenten, griechisch, kyrillisch, arabisch, chinesisch, japanisch etc.) zu schreiben. In Python gibt es Unicode-Strings die dies ermöglichen. Wir treffen Unicode-Zeichenketten vor allem im Zusammenhang mit Umlauten an. Hier nur ein kurzes Beispiel:

```
a = u'Dies ist eine Unicode-Zeichenkette'
# Das u bevor dem Apostroph sagt Python, dass es sich um
# eine Unicode-Zeichenkette handelt

# Unicode-Zeichenketten koennen in eine normale Zeichenkette
# uebersetzt werden. Dafuer benoetigen wir eine
# Zeichenkodierung wie z.B. UTF-8 oder ISO-8859-1
# (uebliche westeuropaeische Kodierung).
b = u'äöü'.encode('utf-8')

# Umgekehrt geht es auch:
c = unicode('äöü', 'iso-8859-1')
```

2.6.3 Code it!

Beantworten Sie die folgenden Fragen:

- Wie viele Wörter gibt es in Shakespeare's Hamlet?
- Wie viele Male hat Shakespeare das Wort 'ghost', 'Ghost', 'GHOST' oder ähnlich geschrieben?

Shakespeare's Hamlet kann von dieser URL als Text-Datei heruntergeladen werden: https://dl.dropbox.com/u/4329479/pg1524.txt.

Und Shakepeare's gesammelte Werke sind als Text-Datei hier verfügbar: https://dl.dropbox.com/u/4329479/pg100.txt.

Für Wundernasen: William Shakepeare hat in seinem Leben 'nur' 5.6 Megabytes an Text geschrieben. Das entspricht in etwa einem Farbbild von rund 1800x1000 Pixel Auflösung!

Eine Text-Datei kann mit Python direkt über das Web in einen String gelesen werden:

```
1 import urllib
2 shake = urllib.urlopen('https://dl.dropbox.com/u/4329479/pg100.
    txt').read()
3 beer = urllib.urlopen('https://dl.dropbox.com/u/4329479/pg1524.
    txt').read()
4 percent_beer = float(len(beer)) / len(shake) * 100
5 print "Hamlet entspricht %f Prozent von Shakespeare's Werken" %
    percent_beer
```

Tip: Versuchen Sie nicht den Inhalt der Variablen *shake* oder *beer* im Terminal auszugeben. Shakespeare's Werke sind doch etwas lang für einen Terminal-Output!

2.6.4 Code it!

Verwandeln Sie folgenden Text in eine Liste von Autopreisen. Jedes Element soll ein Dictionnaire sein; der Key soll der Autoname sein, und der Value der Preis (als Integer). Hier der Text:

`Ford Focus: 25'800, Opel Zafira: 17'900, Subaru Forester: 8'800, BMW 525i: 11'900, Alfa Romeo 156: 4'900`

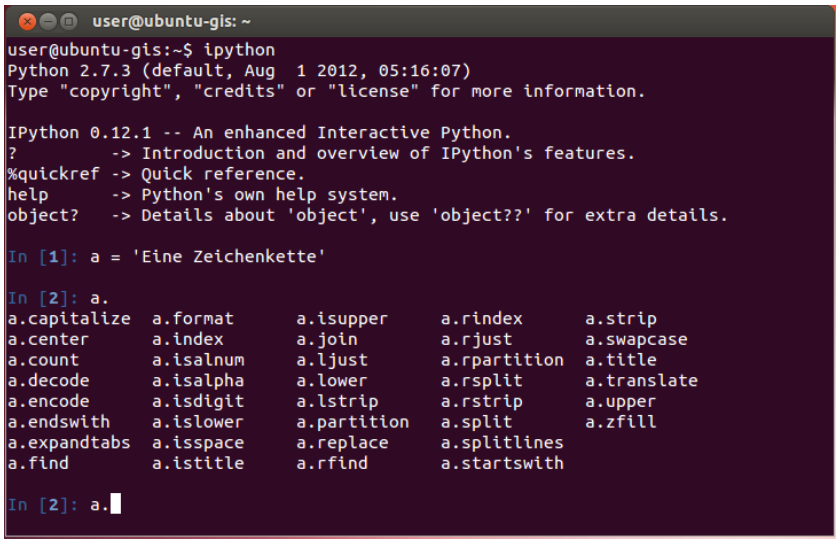

Abbildung 2.1: Alle verfügbaren Funktionen für einen String!

2.7 iPython-Tricks

iPython bietet zwei praktische Funktionen:

- Wir können einen Namen einer Funktion eingeben und anschliessend ein Fragezeichen. Dies gibt uns die Hilfe zu dieser Funktion, falls vorhanden.

- Wir können die ersten Buchstaben einer Funktion eingeben, und anschliessend die Tabulator-Taste drücken. iPython gibt uns dann alle verfügbaren Funktionen aus die mit den eingegebenen Buchstaben anfangen. Dies geht auch zum Beispiel in Zusammenhang mit einer Zeichenkette, die wir in einer Variable gespeichert haben:

```
1    a = 'Eine Zeichenkette'
2    a.      # und anschliessend Tabulator...
```

Figur 2.1 zeigt das Resultat dieser Operation. Wir erhalten so alle verfügbaren Funktionen die auf eine Zeichenkette anwendbar sind!

2.8 Hilfe!

Python hat eine gute Dokumentation, einschliesslich Tutorials und Erklärungen für jede Funktion (Referenz). Die neuste Dokumentation befindet sich hier: http://docs.python.org/2/.

2.9 Kontrolle übernehmen

Programme die einfach von oben nach unten laufen sind ziemlich langweilig. Um den Programmfluss zu beeinflussen brauchen wir zwei Kontroll-Konstrukte: *Schlaufen* und *Bedingungen*. Schlaufen erlauben das mehrfache Ausführen eines Programm-Blocks, während eine Bedingung die Ausführung eines Programm-Blocks nur unter gewissen Umständen ausführt. Wir schauen zwei verschiedene Arten Schlaufen an: **for**- und **while**-Schlaufen, und anschliessend die **if**-Bedingungen.

2.9.1 for-Schlaufen

Eine *for*-Schlaufe führt einen Code-Block für jedes Element einer Liste genau einmal aus (Achtung: das ist anders als meiste Programmiersprachen!). Das sieht dann etwa so aus:

```
1    for a in [1,2,3,4,5]:
2        a2 = a * a
3        print a2
4    print "Schlaufe fertig"
```

Die Schlaufe started also mit *for ... in ...:*. Im ersten Platzhalter definiert man den Namen der Variablen den man dann im folgenden Code-Block benützt um auf den Wert des momentan ausgeführten Listenelements zuzugreifen. Der zweite Platzhalter enthält dann die Liste selbst (oder eine Funktion die eine Liste generiert).

Der Code-Block wird mit dem Einschub (engl. Indentation) abgegrenzt. Für den Einschub können wir einen oder mehrere Tabulatoren oder Leerzeichen verwenden. Üblich sind in Python 4 Leerschläge (Text-Editoren bieten häufig die Option einen Tabulator automatisch in 4 Leerschläge umzuwandeln). Im Beispiel oben ist der mehrfach ausgeführte Code-Block auf Linien 2 und 3.

Achtung: Stellen Sie sicher, dass Sie immer die gleiche Anzahl Leerschläge oder Tabulatoren verwenden, und dass Sie nicht Tabulatoren und Leerschläge vermischen. Sonst gibt es sicher einen Fehler.

Anmerkung: Diese Art Code-Blocks zu definieren ist ziemlich ungewöhnlich. Die meisten Programmiersprachen benützen geschwungene Klammern ({...}) um Code-Blocks abzugrenzen. Der Vorteil von Code-Blocks in Python ist, dass dies besser lesbare Programme gibt.

Das obige Beispiel könnte man etwas einfacher schreiben, da man die Liste `[1,2,3,4,5]` auch automatisch mit einer Funktion generieren kann: `range(1,6)` (6 ist das erste Element das nicht mehr in der Liste ist). Mit **range** können auch 0-basierte Listen erstellt werden, oder Listen mit regelmässigen Schritten:

```
range(5)          # Gibt [0,1,2,3,4]
range(0,10,2)     # Gibt [0,2,4,6,8]
range(5,0,-1)     # Gibt [5,4,3,2,1]
range?            # Gibt die Hilfe zu range in iPython
```

2.9.2 while-Schlaufen

Eine *while*-Schlaufe führt den dazugehörigen Code-Block solange aus bis eine Bedingung nicht mehr erfüllt ist. Hier ein Beispiel:

```
i = 0
while i < 10:
    print i*i
    i += 1        # Addiert 1 zum Wert von i
                  # Identisch zu: i = i + 1
```

Mit einer *while*-Schlaufe ist es einfach möglich, eine unendliche Schlaufe zu programmieren:

```
i = 0
while True:
    print i*i
    i = i % 10 + 1  # Der Modulus-Operator (%) gibt den Rest
       einer Division
```

Drücken Sie Ctrl-C um die Schlaufe zu unterbrechen. Sonst geht sie für immer.

Das scheint auf den ersten Blick keinen Sinn zu machen. Und doch gibt es zusammen mit zwei weiteren Schlüsselwörtern für Schlaufen (for und while-Schlaufen) viele Anwendungsmöglichkeiten:

- **continue**: Bricht die Ausführung des Schlaufen-Code-Blocks ab und macht mit dem nächsten Schlaufen-Durchgang weiter (also oben am Code-Block).
- **break**: Bricht wie *continue* die Ausführung des Schlaufen-Code-Blocks ab, beendet jedoch auch gleich die Schlaufe.

2.9.3 if-Bedingungen

Eine *if*-Bedingung führt den dazugehörenden Code-Block nur unter gewissen Bedingungen aus. Das kann dann so aussehen:

```
1  import random   # Um Zufallszahlen zu generieren
2  while True:     # Eine unendliche Geschichte
3      i = random.randint(0,100)       # Eine Zufallszahl
           zwischen 0 und 100
4      if i < 50:
5          print 'i ist %i' % i
6      if i == 33:     # == vergleicht zwei Werte miteinander
7          break
```

Bedingungen können auch kombiniert werden (aber die zu vergleichende Variable muss immer wiederholt werden):

```
1  import random
2  while True:
3      i = random.randint(0,100)
4      if i < 20 or i == 35 or i >= 80:
5          print 'i ist %i' % i
6      if i == 33:
7          break
```

In einer Bedingung wird einfach geschaut ob der Wert nach dem `if`-Schlüsselwort `True` oder `False` ist, und wenn es `True` ist wird der Code-Block ausgeführt. Dabei kann auch durchaus eine Variable `True` oder `False`

enthalten, oder eine Funktion einen der Werte zurückgeben. Es gibt viele verschiedene Varianten zum Testen:

- **Gleichheit**: Mit `==` oder `is` testen. Also `i == 3` oder `i is 3`.
- **Ungleichheit**: Mit `!=` oder `is not` testen.
- **Grösser als**: Mit `>` oder `>=` für grösser oder gleich.
- **Kleiner als**: Mit `<` oder `<=` für kleiner oder gleich.

Es können auch *else*-Anweisungen und *else if*-Anweisungen (`elif` genannt in Python) ausgeführt werden:

```
1    mein_name = 'Hans'
2    if mein_name == 'Peter':
3        print 'Hallo Peter, wie geht es Dir?'
4    elif mein_name == 'Petra':
5        print 'Hallo Petra'
6    else:
7        print 'Hallo Unbekannter'
```

2.10 Funktionen definieren

Funktionen sind wichtige Elemente im Programmieren. Sie erlauben es grosse Anteile an Code auszulagern. Das hat zwei Vorteile: einerseits wird unser 'Hauptcode' viel lesbarer und kürzer, und zweitens können wir eine Funktion woanders wieder verwenden. Eine Funktion hat einen eindeutigen Namen, man kann zwischen 0 und N Variablen beifügen die die Funktion dann verwenden kann, und die Funktion kann 0 oder 1 Variable als Resultat zurückgeben. Eine Funktion wird mit dem Schlüsselwort *def* definiert, die Argumente werden in Klammern übergeben, und ein eventuelles Resultat wird mit *return* zurückgeschickt:

```
1    def meine_funktion(argument1, argument2):
2        print argument1
3        return argument2
```

Die Funktion kann dann einfach so ausgeführt werden:

```
    resultat = meine_funktion('hello', 'world')
```

Variablennamen innerhalb einer Funktion sind ausserhalb der Funktion nicht gültig, und umgekehrt. Im folgenden Beispiel wird `5` ausgegeben, und nicht etwa `3`:

```
    a = 5
    def eine_funktion():
        a = 3
        return a
    b = eine_funktion()      # b ist 3
    print a      # a ist immer noch 5
```

In Python ist es auch möglich, optionale Argumente zu definieren. Diese müssen jedoch immer am Schluss stehen. Dabei wird einfach ein Default-Wert für das optionale Argument definiert:

```
    def addition(a, b=1):
        return a + b

    print addition(3, 5)     # Gibt 8
    print addition(3)        # Gibt 4
```

Es ist auch möglich die Argumente im Funktionsaufruf beim Namen zu nennen und so Argumente zu überspringen oder deren Ordnung zu ändern. Allerdings müssen in jedem Fall alle obligatorischen Argumente vorhanden sein:

```
    def addition(a, b=1, c=2):
        return a + b + c

    print addition(a=1)          # Gibt 4 (1 + 1 + 2)
    print addition(2, c=4)       # Gibt 7 (2 + 1 + 4)
    print addition(c=1, b=1, a=0)   # Gibt 2
    print addition(3, 1, c=0)    # Gibt 4
    print addition(3, 2, b=4)    # Fehler da b doppelt definiert
    print addition(a=1, 2)  # Fehler da 2. Argument ohne Namen
```

Aufpassen muss man auch, dass in einer Funktion die Werte der Argumente verändert werden können:

```
def liste_erweitern(liste):
    liste.append(1)     # Haengt 1 an die Liste

eine_liste = [1,2,3]
liste_erweitern(eine_liste) # Liste ist nun [1,2,3,1]
```

Aber in diesem Fall wird das Argument nicht verändert:

```
def liste_erweitern(liste):
    liste = liste + [1] # Haengt 1 an die Liste
        # liste ist nun eine neue Variable
        # Wir haetten ja auch durchaus liste2 schreiben
        # koennen ohne das Verhalten zu aendern
    return liste

eine_liste = [1,2,3]
eine_neue_liste = liste_erweitern(eine_liste)
    # eine_liste bleibt unveraendert
    # eine_neue_liste enthaelt [1,2,3,1]
```

Die Namensgebung der Funktionen ist relativ frei, und ist gleich wie bei Variablen. Eine Funktion muss mit einem Buchstaben oder 'Underscore' (_) beginnen, und darf dann auch Zahlen enthalten. Gross-/Kleinschreibung wird berücksichtigt. Funktionen und Variablen die mit zwei Underscores anfangen und aufhören sind für die interne Funktionalität von Python reserviert. Achten muss man jedoch bei der Namensgebung von Funktionen, dass sie nicht den gleichen Namen wie Variablen haben und umgekehrt. Funktionen sind nämlich auch Variablen! Und zwar sind es Variablen die den Datentyp 'Funktion' haben... Also folgender Code funktioniert einwandfrei:

```
def quadrat(a)
    return a*a

au_carre = quadrat
quadrat = 'Das ist unnuetz und total verwirrend!'
print au_carre(4)   # Gibt 16
type(au_carre)      # Gibt den Typ der Variable au_carre aus:
    eine Funktion!
```

2.11 Operatoren

Wir haben schon einige Operatoren gesehen. Hier noch eine Auflistung der meist gebrauchten Operatoren:

- `+ - * /`: die gängigen arithmetischen Operatoren.
- `**`: 'hoch'. Also zum Beispiel `2**3` $= 2^3$
- `%`: 'modulus'. Der Rest einer Division. Also z.B. `7 % 4` gibt 3.

Kapitel 3

Daten lesen und schreiben

3.1 Einfaches Lesen und Schreiben

Die Funktion `open` öffnet eine Datei und gibt ein File-Objekt das es uns erlaubt, Operationen auf die offene Datei durchzuführen. Die Funktion `open` verlangt den Datei-Pfad und optional den Modus in dem die Datei geöffnet werden soll. Die möglichen Optionen sind dabei '`r`' (default, nur Leseoperationen sind erlaubt), '`w`' (nur Schreiben erlaubt), '`a`' (append; am Ende der Datei hinzufügen) und '`r+`'. Beispiel um eine Datei im Nur-Lese-Modus zu öffnen:

```
f = open('meine-datei.txt')
```

Falls die Datei '`meine-datei.txt`' nicht existiert wird im '`w`'-Modus eine neue Datei mit diesem Namen erstellt. Eine inexistente Datei in einem anderen Modus zu öffnen ist ein Fehler.

Achtung! Wenn wir eine existierende Datei im Schreib-Modus öffnen wird der existierende Inhalt überschrieben! Und zwar ohne Rückfrage oder Warnung!

Um den Inhalt der Datei zu lesen gibt es mehrere Möglichkeiten:

```
s = f.read()     # Liest die ganze Datei in die Variable s
                 # Aufpassen mit grossen Dateien! Die Datei muss
                 # in den Arbeitsspeicher passen.
s = f.readline()     # Liest die naechste Linie in die Variable s
s = f.readlines()    # Liest die ganze Datei in eine Liste,
                     # wo jedes Listenelement einer Linie
```

```
7                       # entspricht
```

Falls `readline()` das Ende der Datei erreicht, wird ein leerer String (”) zurückgegeben, im Gegensatz zu einer leeren Linie, wo immer das Zeilenende-Zeichen zurückgegeben wird. Das Zeilenende-Zeichen ist das Zeichen ’`\n`’ auf Unix-Systemen und ’`\r\n`‘ (also 2 Zeichen!) auf Windows-Systemen (dieser Unterschied ist übrigens auch der Grund warum Unix-Textdateien als eine einzige Linie in Windows erscheinen).

Um eine Datei komplett von Anfang bis Ende Zeile um Zeile zu lesen, können wir einfach eine `for`-Schlaufe benutzen:

```
1     f = open('meine-datei.txt')
2     for s in f:
3         # Etwas mit s machen...
4         print s
```

Eine offene Datei enthält einen Datei-Cursor. Dieser Cursor merkt sich, wo wir momentan in der Datei sind mit dem Lesevorgang. Dies ermöglicht es uns überhaupt erst, die Datei stückweise zu lesen. Aber dies bedeutet auch, dass wir nicht rückwärts lesen können. Um dies zu ermöglichen, müssen wir den Datei-Cursor manuell zurücksetzen. Dies macht die Funktion `seek()`, die als Argument die Anzahl Bytes ab Dateianfang braucht, um den Datei-Cursor an der entsprechenden Stelle zu platzieren. Um den Cursor an den Datei-Anfang zu setzen können wir also einfach schreiben `f.seek(0)`.

Brauchen wir die Datei nicht mehr in unserem Programm, müssen wir die Datei wieder schliessen mit `f.close()`.

Das Schreiben einer Datei ist auch nicht kompliziert:

```
1     import os, sys
2     a = 45.6
3     # Zuerst Test machen ob Datei schon existiert.
4     #Damit wollen wir vermeiden, dass wir eine existierende
5     # Datei stillschweigend ueberschreiben.
6     if os.path.exists('meine-datei.txt'):
7         print 'Datei existiert schon!'
8         sys.exit(0) # Programm stoppen
9     f = open('meine-datei.txt', 'w')
10    f.write('Erste Zeile der Datei.\n')
11    f.write('Der Wert von a ist %f\n' % a)
```

```
12      f.close()
```

3.2 Python-Objekte speichern

Ganze Python-Objekte können ganz einfach in Binär-Dateien geschrieben und wieder gelesen werden. Das geht mit dem Modul `pickle` oder `cPickle` (die Funktionalität ist praktisch identisch). Hier ein Beispiel:

```
1   import pickle
2   d = {'Antoine': 45, 'Berta': 34, 'Claudio': 23}
3   f = open('abc.pickle', 'w')
4   pickle.dump(d, f)
5   f.close()
```

Und das Lesen geht dann so:

```
1   import pickle
2   f = open('abc.pickle')
3   d = pickle.load(f)
4   f.close()
```

Anstatt in eine Datei zu schreiben können wir das Python-Objekt auch in einen String 'schreiben' und auch wieder 'einlesen':

```
1   import pickle
2   d = {'Antoine': 45, 'Berta': 34, 'Claudio': 23}
3   d_string = pickle.dumps(d)
4   d_neu = pickle.loads(d_string)
```

3.3 JSON-Dateien lesen und schreiben

JSON ist die Abkürzung für 'JavaScript Object Notation'. Es ist ein einfache Text-basierte Repräsentation für Zahlen, Strings, Listen und Dictionnaires (in JavaScript können Dictionnaires wie Objekte behandelt werden und umgekehrt). Das interessante an JSON ist, dass es gleichzeitig der Syntax von

Python und Javascript entspricht. Ein Beispiel für ein Punkt-Feature wie wir es aus Geographischen Informationssystemen kennen:

```
{
    "type": "Feature",
    "properties": {"id": 1, "name": "Bern"},
    "geometry": {
        "type": "Point",
        "coordinates": [600000.0, 200000.0]
    }
}
```

Dieser Code kann automatisch von Python gelesen und geschrieben werden, und zwar mit dem Modul `json`:

```
import json
pt = {
    "type": "Feature",
    "properties": {"id": 1, "name": "Bern"},
    "geometry": {
        "type": "Point",
        "coordinates": [600000.0, 200000.0]
    }
}
f = open('mein-feature.json', 'w')
json.dump(pt, f)
f.close()
```

Und zum Lesen:

```
import json
f = open('mein-feature.json')
pt = json.load(f)
f.close()
```

3.4 Daten aus URLs lesen

Eine URL kann ähnlich zu einer Datei gelesen werden (schreiben geht natürlich nicht). Dafür verwenden wir das Modul `urllib`. Hier ein Beispiel:

```
import urllib
url = 'http://transport.opendata.ch/v1/connections?from=
    Rapperswil&to=Chur&date=20120913&time=1800'
    # Fahrplan-Anfrage von Rapperswil nach Chur
    # am 13. September 2012 um 18:00 Uhr
url_connection = urllib.urlopen(url)
content = url_connection.read()
    # Wir bekommen JSON zurueck...
import json
result = json.loads(content)
erste_verbindung = result['connections'][0]
abfahrt = erste_verbindung['from']['departure']
ankunft = erste_verbindung['to']['arrival']
...
```

Wir haben also eine Funktion `urlopen` die als Argument die URL braucht, und danach können wir den Inhalt der URL ganz einfach mit der `read()`-Funktion in einen String lesen. Anschliessend können die gewünschten Informationen aus dem Resultat gefiltert werden. Im obigen Beispiel haben wir JSON-Daten gelesen die wir in ein Python-Dictionnaire verwandeln können. Je nachdem bekommen Sie natürlich auch anderen Inhalt zurück, zum Beispiel HTML-Dateien die Sie dann analysieren können und so ihre persönliche Suchmaschine basteln (wie das im Detail geht erfahren Sie auf udacity.com).

Kapitel 4

Einführung in Python-Module

4.1 Was ist ein Modul?

Ein Modul ist eine Code-Bibliothek, die Funktionen, Variablen und Klassen (werden wir später kennenlernen) enthält. Ein Modul wird in Python mit der *import*-Anweisung in unser Programm geladen:

```
1    import math # Laedt das Modul math das standardmaessig in
        Python enthalten ist
```

Das Modul `math` enthält unter anderem eine Variable `pi` und eine Funktion `sin`. Für den Aufruf dieser Variablen und Funktionen müssen wir den Namen des Moduls angeben:

```
1    import math
2    print math.sin(3*math.pi/2)
```

Nachdem das Modul `math` geladen wurde, befinden sich die Funktionen und Variablen innerhalb des *'Namensraums'* `math`. Deshalb müssen wir diesen Namen jedes Mal angeben um eine Funktion oder Variable aus diesem Namensraum zu benützen. Dies dient im Wesentlichen dazu, dass existierende Variablen und Funktionen nicht einfach überschrieben werden durch den Inhalt eines Moduls; dies wäre eine potentielle obskure Fehlerquelle! Es ist jedoch möglich, beim Laden eines Moduls den Namensraum zu definieren, also zum Beispiel:

```
1    import math as m
```

```
2     print m.sin(3*m.pi/2)
```

Auch hier gilt wieder: ein Modul ist wie eine Funktion ein separater Datentyp, und `m` ist in diesem Fall wie eine Variablennamen zu behandeln. Dementsprechend funktioniert der folgende Code nicht:

```
1     import math as m
2     m = 123
3     print m.sin(3*m.pi/2)
```

Jedoch dieser Code funktioniert einwandfrei (auch wenn diese Praxis nicht zu empfehlen ist):

```
1     import math as m
2     n = m
3     m = 123
4     print n.sin(3*n.pi/2)
```

Es ist auch möglich nur gewisse Funktionen oder Variablen aus einem Modul zu laden (und dies ist eine gute Praxis da wir nur das importieren was wir auch brauchen):

```
1     from math import sin, pi
2     print sin(3*pi/2)
```

Zu beachten ist hier, dass dabei kein separater Namensraum für das Modul `math` erstellt wird, und `sin` und `pi` direkt benützt werden können. Es ist auch möglich, aber nicht empfehlenswert, alle Funktionen und Variablen zu importieren ohne einen neuen Namensraum zu erstellen:

```
1     from math import *
2     print sin(3*pi/2)
```

4.2 Ein eigenes Modul erstellen

Es ist sehr einfach, ein eigenes Python-Modul zu erstellen. Im Grunde genommen ist jede Skript-Datei mit Endung `.py` ein Python-Modul. Erstellen wir

also probehalber ein neues Python-Modul. Wir erstellen eine neue Text-Datei mit Namen `hello.py`, die wir im aktuellen Working Directory speichern. Wir schreiben folgenden Code in die Datei:

```
1    def hello_world():
2        print 'Hello World!'
```

Nun können wir dieses Python-Modul in der Python-Konsole importieren:

```
1    import hello          # Der Datei-Name ohne die Endung
2    hello.hello_world()
```

Nach dem erstmaligen Importieren des Moduls werden Sie eine neue Datei finden, die den Namen `hello.pyc` hat. Diese Datei enthält kompilierten Python-Code. Python kompiliert nämlich die Module, damit die Ausführung des Codes schneller geht. Wir brauchen uns jedoch nicht um diese Kompilierung zu kümmern, die geht automatisch vonstatten, auch wenn wir Änderungen im Modul vornehmen (allerdings müssen wir dann das Modul neu importieren mit `reload(...)`.

Python-Module können auch komplexer sein und Sub-Module enthalten. Irgendein Ordner der eine Datei mit Namen '`__init__.py`' (mit zwei Underscores) gilt in Python als Modul. Der Ordnername wird dabei als Modulname verwendet.

4.3 Ein Modul installieren: der PYTHONPATH

Nur mit dem Erstellen eines Moduls ist es jedoch nicht automatisch in jedem Skript verfügbar. Wenn wir mit `import` versuchen ein Modul zu importieren, such Python nämlich nur in ganz bestimmten Ordnern ob es ein Modul mit dem entsprechenden Namen finden kann. Die Liste mit diesen Ordnern wird der *PYTHONPATH* genannt. Der Python-Path ist eine normale Python-Liste mit Pfaden zu Ordnern. Diese Liste befindet sich in der Variable `path` im `sys`-Modul. Der Inhalt wird beim Aufstarten von Python automatisch generiert, aber wir können durchaus einen neuen Ordner hinzufügen:

```
1    import sys
2    sys.path.append('/home/user/meine-python-module')
```

Von nun an können wir unsere Python-Module in den Ordner 'meine-python-module' legen und Python findet diese Module automatisch. Jedoch müssen wir diese Operation beim nächsten Aufstarten von Python wiederholen, was ziemlich unpraktisch ist. Um ein Modul permanent in unser Python zu installieren haben wir zwei Möglichkeiten:

- Wir legen unser Modul ganz einfach in einen Ordner der schon im Python-Path steht.
- Wir erstellen unseren eigenen Python-Modul-Ordner (*/home/user/meine-python-module*). In einem der Ordner der schon im Python-Path steht erstellen wir eine Datei mit Endung *.pth*, also z.B. *meine-python-module.pth*. Diese Datei ist eine einfache Text-Datei, wo wir ganz einfach den Pfad zu unserem Modul-Ordner einfügen (also */home/user/meine-python-module/*).

Durch diesen Mechanismus ist Python eine leicht erweiterbare Programmiersprache. Neue Module können einfach installiert werden. Einige Module sind jedoch auch in anderen Programmiersprachen wie z.B. C oder Fortran geschrieben. Diese Module müssen vor der Installation kompiliert werden. Dafür gibt es spezielle Installationstools, wie zum Beispiel die `setuptools`, das selbst ein Python-Modul ist. Unter Linux kann auch der Package-Manager standardmässig viele gängige Python-Module installieren. Z.B. das Python-Modul `numpy` in Ubuntu zu installieren geht ganz einfach:

```
1    sudo apt-get install python-numpy
```

Für Windows gibt es meist Installationsprogramme, die meist einfach zu handhaben sind.

Für Mac OS X empfiehlt es sich einen Package-Manager wie zum Beispiel MacPorts zu installieren, und die Installation geht dann ähnlich wie unter Ubuntu:

```
1    sudo port install py27-numpy
```

4.3.1 setuptools

`setuptools` ist ein optionales Python-Modul das das Installieren und Verteilen von Python-Modulen vereinfachen will. Es erlaubt uns ein sich in einem

Ordner befindenden Modul einfach zu installieren durch die Ausführung des 'setup.py' Skripts: `python setup.py install`

4.4 Ein paar nützliche eingebaute Module

4.4.1 sys

Das Modul `sys` enthält einige Basis-Funktionen und -Variablen. Dabei haben wir den `sys.path` schon gesehen. `sys` enthält auch eine Variable `platform` die aussagt auf welchem Betriebssystem wir gerade sind. Im Modul`sys` hat es auch eine Funktion `exit()` die das Python-Programm oder den Interpreter beendet.

4.4.2 os

Das Modul `os` ist ein Interface zum Betriebssystem (Operating System). So können wir mit `os.getcwd` den aktuellen Arbeitsordner (get current working directory) abfragen, oder ihn mit `os.chdir` ändern. Die Variable `os.sep` enthält das Zeichen das zwischen zwei Ordnern steht in einem Pfad. Das ist für Windows der Backslash (\), und für Unix-basierte Systeme der normale Slash (/).

`os` enthält auch Funktionen um einen neuen Ordner zu erstellen (`mkdir()`), oder zu entfernen (`rmdir()`).

4.4.3 shutil

Das Modul `shutil` gibt uns Zugriff auf Funktionen die man typischerweise in der Unix-Konsole findet. Unter anderem findet man da folgende Funktionen:

- `copytree()` um einen Ordner und all seinem Inhalt zu kopieren.
- `rmtree()` um einen Ordner mitsamt seinem Inhalt zu löschen. Logischerweise muss man da **extrem vorsichtig sein**. Eine falsche Manipulation kann nicht rückgängig gemacht werden (ausser mit einem sehr guten Backup).
- `abspath()` wandelt einen relativen Pfad (zum Beispiel '../../lib') in einen absoluten Pfad um (also zum Beispiel '/lib').

- `move()` erlaubt es einen Ordner oder eine Datei an einen anderen Ort zu verschieben.

4.4.4 math

Wir haben das Modul `math` schon ansatzweise gesehen. Es enthält viele gängige mathematische Funktionen und Konstanten:

- `e` und `pi` zwei berühmte Konstanten.
- `ceil()` rundet ein `float` auf (bleibt aber ein `float`). Um ein `float` auf den nächst höheren `int` aufzurunden: `int(math.ceil(2.3))`.
- `floor()` rund ein `float` ab (bleibt aber auch ein `float`).
- `log()` und `log10()` rechnen den Logarithmus, und `exp()` die Exponentialfunktion (also `exp(5)` $= e^5$).
- `sqrt()` ist die Quadratwurzel. Also `math.sqrt(4)` ist das gleiche wie `4**0.5`.
- Die trigonometrischen Funktionen wie `sin()`, `cos()`, `asin()` usw. sind auch vorhanden.

Die komplette Dokumentation für das `math`-Modul kann an der Adresse http://docs.python.org/library/math.html gefunden werden.

4.4.5 random

Das Modul `random` enthält Funktionen für die Generierung von Pseudo-Zufallszahlen. Die nützlichsten Funktionen sind:

- `random()` gibt ein zufälliger `float` zwischen 0 und 1 zurück (0 eingeschlossen, jedoch ohne 1).
- `randint()` gibt ein zufälliger `int` zurück. Das erlaubte Intervall muss angegeben werden, also z.B. `randint(0,5)` gibt ein `int` der gleich 0,1,2,3,4 oder 5 ist.
- `choice()` gibt ein zufälliges Element einer Liste zurück. Wir können also `randint(0,5)` auch durch `choice(range(6))` ersetzen.

4.4.6 datetime

Das `datetime`-Modul erlaubt es mit Datum und Zeit umzugehen. Hier ein paar nützliche Beispiele:

```
1  import datetime
2
3  heute = datetime.date.today()
4  print heute.year, heute.month, heute.day
5
6  jetzt = datetime.datetime.now()
7  print jetzt.day, jetzt.hour, jetzt.minute
8
9  feiertag = datetime.date(2012, 8, 1)
```

Das `time`-Modul gibt ein paar zusätzliche nützliche Funktionen:

- `time.time()` gibt die Anzahl Sekunden seit dem 1.1.1970. Wenn wir das am Anfang und am Ende unseres Skripts anfragen können wir schauen wie lange unser Skript gedauert hat.
- `time.sleep()` erlaubt es unseren Code ein paar Sekunden "schlafen" zu lassen.

Die ausführliche Dokumentation zu diesen Modulen befindet sich an folgenden Adressen:

- http://docs.python.org/library/datetime.html
- http://docs.python.org/library/time.html

4.4.7 re

`re` steht für 'Regular Expressions'. Regular Expressions (RegEx) sind eigentlich ein Thema das in einem separaten Kurs behandelt werden sollte. Wir werden deshalb nicht die Regular Expressions an sich behandeln.

Regular Expressions erlauben es nach komplexen Patterns in einem Text zu suchen, und auch zu ersetzen. Regular Expressions sind nicht spezifisch zu Python, sondern eine Art Sprache für sich. In meisten Programmiersprachen gibt es eine Library für Regular Expressions. Deshalb hier nur ein kurzes Beispiel:

```
1  import re
2  m = re.search('([0-9]+),(.*?),', '134,Hello world!,Yeah!')
3  # Suche nach 1-N Zahlen gefolgt von einem Komma und
4  # dann 0-N beliebige Zeichen bis zum naechsten Komma
5  m.group(1)  # Enthaelt die erste Gruppe (134)
6  m.group(2)  # Enthaelt die zweite Gruppe (Hello world!)
```

Die Dokumentation zu `re` befindet sich hier: http://docs.python.org/library/re.html. Es gibt zahlreiche Tutorials zu Regular Expressions, ein empfehlenswertes ist hier: http://code.google.com/edu/languages/google-python-class/regular-expressions.html

Kapitel 5

Programme strukturieren

In diesem Kapitel wollen wir versuchen, ein paar Richtlinien und Tipps zusammenzustellen, die helfen sollen, gut strukturierte Programme zu schreiben. Ein gut strukturiertes Programm ist einfacher zu verstehen und den Code zu lesen und zu schreiben. Dadurch ist es auch einfacher das Programm zu testen, und schlussendlich ist es auch stabiler und zuverlässiger.

Zuerst fangen wir an, ein Template für ein typisches Python-Projekt zu bauen, das wir dann für alle Python-Projekte verwenden können. Anschliessend schauen wir noch das Problem an, wie wir den Weg von einer Programm-Idee zu einem funktionierenden Programm finden können.

5.1 Ein typisches Python-Projekt

Ein typisches Python-Projekt sollte in etwa die in Figur 5.1 gezeigte Struktur haben. Dabei ist `NAME` der Name von unserem Python-Modul das wir erstellen wollen. Dieser Name muss natürlich für jedes Projekt dann separat geändert werden.

Aus dieser Struktur können wir folgende Schlüsse ziehen:

- Ein Python-Projekt braucht immer Dokumentation (docs-Ordner). Wirklich: ein Python-Projekt braucht immer Dokumentation. Wirklich.

- Um die Installation zu vereinfachen erstellen wir eine `setup.py`-Datei. Damit können wir unser Python-Modul auch unseren Freunden verteilen.

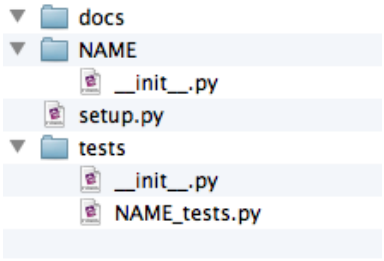

Abbildung 5.1: Struktur eines typischen Python-Projekts.

- Ein Python-Projekt braucht immer Tests (tests-Ordner). Wirklich: ein Python-Projekt braucht immer Tests. Wirklich.

Die beiden `__init__.py`-Dateien sind leer und existieren bloss um den Ordner NAME bzw. tests in Python-Module zu verwandeln. Unser Modul-Code kann dann in die `NAME/__init__.py`-Datei gehen, oder in eine separate Datei mit beliebigen Namen.

Die `setup.py`-Datei enthält Informationen zu unserem Projekt und den Code, der für eine Installation mit dem Befehl `python setup.py install` notwendig ist:

```
try:
    from setuptools import setup
except ImportError:
    from distutils.core import setup

config = {
    "name": "NAME",
    "description": "Mein Python-Modul",
    "url": "http://url.wo/mein/projekt/beschrieben/ist",
    "download_url": "http://url.wo/mein/modul/heruntergeladen/
       werden/kann.zip",
    "author": "ICH",
    "author_email": "ich@i.ch",
    "version": "0.0.1",
    "install_requires": ["nose"],
    "packages": ["NAME"],
    "scripts": [],
}

```

```
19 setup(**config)
```

Die Datei `tests/NAME_tests.py` enthält Code der unser Modul auf die Funktionstüchtigkeit testet:

```
1  import nose.tools
2  import NAME
3
4  def setup():
5      print "setup"
6
7  def teardown():
8      print "teardown"
9
10 def test_irgendetwas():
11     print "Irgend ein Test ist gemacht"
```

Dabei verwenden wir das optionale Python-Modul `nose` das spezifisch für das Testen von Python-Code erstellt wurde. Falls `nose` auf dem Computer installiert ist, können wir im Terminal innerhalb dem NAME-Ordner den Befehl `nosetests` ausführen. Dabei werden die Tests automatisch durchgeführt.

Selbstverständlich müssen wir in der `test_irgendetwas`-Funktion (oder irgendeiner `test_xxx`-Funktion) auch wirklich etwas nützliches testen...

5.2 Von der Idee zum Code

Ein häufiges Problem ist, dass wir eine relative vage Idee im Kopf haben, aber nicht recht wissen wie wir diese Idee umsetzen können. Ein guter Ansatz ist hier sich ein kleines aber konkretes Ziel zu setzen das in die richtige Richtung zu gehen scheint. Anschliessend, wenn wir dieses erste Ziel erreicht haben, können wir iterativ neue kleine Ziele setzen die auf dem Erreichten aufbauen. Wenn wir im Stande sind, eine Idee in kleinere Teile zu zerlegen, gibt uns das häufig auch gleich Ansätze zur Programmstruktur. Grundsätzlich sollten wir den Code in separate Teile zerlegen, wo die einzelnen Probleme voneinander getrennt und wenn möglich unabhängig gelöst werden können. Diese einzelnen Teile können dann zum Beispiel in eine Funktion oder in ein separates

Modul integriert werden. Manchmal braucht es auch verschiedene Ansätze gleichzeitig.

Es ist auch wichtig zu wissen, dass ein Programm nicht am Computer entsteht, sondern zuerst als Idee im Kopf, und dann als Skizze auf Papier. Es ist wichtig eine präzise Idee zu haben von dem was man erreichen will bevor man sich an den Computer setzt. Dabei kann jede Art Skizze, Flussdiagramm oder Pseudo-Code behilflich sein. Wir müssen in dieser Phase unbedingt zwei Dinge erreichen: 1. muss das Ziel klar sein und 2. muss das Problem genug klein sein um es auch lösen zu können. Falls einer dieser Punkte nicht klar ist, werden wir beim Programmieren scheitern.

Machen wir ein kleines Beispiel. Wir haben eine Adress-Datei im Text-Format, und wir wollen für jede der Adressen die Koordinaten finden und eine Datei erstellen, die ich anschliessend mit einem Geographischen Informationssystem (GIS) wie z.B. QuantumGIS (qgis.org) einlesen, anzeigen und abfragen kann. Um die Koordinaten für die einzelnen Adressen zu finden verwenden wir einen WebService. Dieses Problem ist zugegebenermassen klein und überschaubar. Trotzdem können wir das Problem zum Beispiel wie in Figur 5.2 skizzieren. Diese einfache Skizze erlaubt es uns die verschiedenen Teile des Programms zu sehen und separat zu programmieren und testen. Wir könnten zum Beispiel zuerst am Schluss des Programms anfangen und eine Funktion schreiben, die beliebige Adressen mit zugehörigen Punkt-Koordinaten in eine Datei schreibt, die ich dann mit einem GIS einlesen kann. Dazu muss man sich sicher fragen, welches Datenformat man wählen will, ob man dieses Datenformat schreiben kann, und wie man die geokodierten Adressen in Python-Datenstrukturen speichern will. Man könnte zum Beispiel eine simple Textdatei als Format wählen. Eine Alternative ist auch GeoJson, das in Python sehr einfach zu lesen und schreiben ist dank dem Modul `json`, und das mit QuantumGIS einfach eingelesen werden kann. Nun haben wir ein kleines Problem das wir mal versuchen können umzusetzen. Dabei können wir selbstverständlich auch nach schon existierenden Beispielen suchen, die wir dann nutzen können sofern erlaubt. Als Resultat könnten wir hier zum Beispiel ein Dictionnaire erstellen, der alle Daten einer Adresse inklusive Koordinaten speichert. Weiter brauchen wir dann eine Funktion `as_json(adressen)`, die eine Liste mit Adressen in eine GeoJson-Datei umwandelt. Die Datenstruktur (der Dictionnaire) und die Funktion können auch in anderen Programmen verwendet werden, also werden wir sie in ein sinnvolles Modul legen. Wir können auch Code schreiben, der diese Elemente testet. Wenn dies erledigt ist, können wir uns dem nächsten Teilproblem-

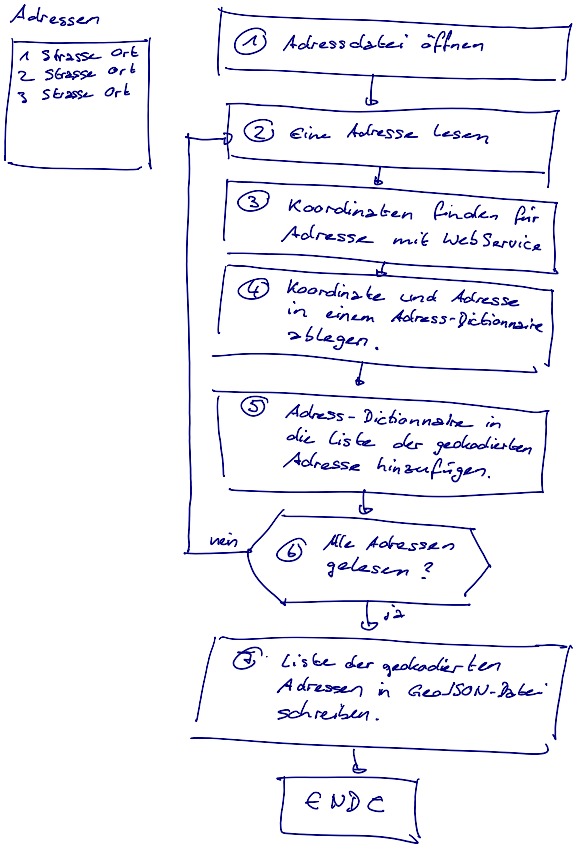

Abbildung 5.2: Einfache Skizze eines Geokodier-Programmes.

chen widmen und so unserem fertigen Programm Schritt für Schritt näher kommen.

Nebst Skizzen und Notizen auf Papier ist es auch wichtig den Code gut zu dokumentieren. Dafür offeriert Python das Konzept des 'Doc-String', der da ist um den Programmierer zu informieren für was ein Modul oder eine Funktion gut sein soll. Der Doc-String steht immer unmittelbar nach der Definition der Funktion, oder ganz oben in einem Modul. Zum Beispiel könnten wir ein Modul `geocoding` erstellen das unsere `as_json`-Funktion enthält. Die `geocoding.py`-Datei könnte dann so aussehen:

```
1 """Das geocoding-Modul enthaelt Datenstrukturen und Funktionen
2    die es erlauben Adressen zu geokodieren und das Resultat in
```

```
    eine GIS-kompatible Datei zu schreiben.    ...
"""

bsp_adresse = {
    'strasse': 'Beverinstrasse 2',
    'plz': 7430, 'ort': 'Thusis', 'land': 'Schweiz',
    'coords': [46.693565, 9.434898]
}

def as_json(adressen):
    """Verwandelt eine Liste mit Adress-Dictionnairen ins
    GeoJson-Format.
    """
    ...
```

Nutzen Sie diese Möglichkeit ihren Code so zu dokumentieren, zusätzlich zu den normalen Kommentaren (die selbstverständlich auch notwendig sind). Dokumentieren Sie Ihren Code so, dass auch Ihre Grossmutter versteht was der Code machen soll. Dann haben Sie eine Chance den Code auch in 2 Monaten noch zu verstehen...

Hier noch ein paar hilfreiche Tipps für die Programm-Strukturierung:

- Teilen Sie das Problem in mehrere kleine Probleme auf, die Sie auch lösen können.
- Versuchen Sie das Programm in verschiedene logische Einheiten zu zerlegen, die möglichst unabhängig sind voneinander. Packen Sie diese logische Einheiten zusammen in ein Modul. Der Algorithmus der die Koordinaten findet für eine Adresse muss nicht wissen was mit den Koordinaten anschliessend passiert. Und er muss auch nicht wissen woher die Adresse kommt.
- Versuchen Sie die logischen Einheiten so generell wie möglich zu schreiben. Damit können Sie diese Teile in einem anderen Programm wieder verwenden und eventuell erweitern. Ein Modul zum Beispiel sollte jedoch nur so viel Code wie nötig enthalten, also so wenig wie möglich.
- Schreiben Sie den Code in einer Art, dass er logisch aufgebaut ist, einfach lesbar und verständlich. Brauchen Sie Kommentare falls nötig. Jedoch: wenn Sie Ihren Code um eine Linie verkürzen können ohne das Verständnis zu erschweren, tun Sie es.

- Testen Sie jeden Teil separat auf Fehler. Normalerweise ist es nötig ein separates Programm zu schreiben das als einziges Ziel hat den Programmteil zu testen.
- Zögern Sie nicht Skizzen und Notizen auf Papier zu machen. Das hilft oft ungemein beim Programmieren.
- Nehmen Sie sich Zeit um zu überlegen wie Sie Ihr Programm strukturieren wollen. Wenn Sie einen Kollegen haben der auch programmiert, diskutieren Sie mit ihm darüber. Evaluieren Sie sorgfältig verschiedene Optionen. Zum Beispiel für das GIS-kompatible Dateiformat im obigen Beispiel, schauen Sie welche Formate die GIS-Software lesen kann. Versuchen Sie eventuell Bibliotheken zu finden die solche Dateien schreiben können. Evaluieren Sie solche Bibliotheken: sind sie einfach zu benutzen? Wie ausgereift sind die Bibliotheken? Wie gut ist die Dokumentation? Häufig gibt es Beispiele die mit der Bibliothek daher kommen. Probieren Sie diese Beispiele aus! Somit können Sie abschätzen ob die Bibliothek ihnen auch entspricht. Im Falle der GIS-Vektorformate ist das GeoJson-Format am einfachsten zu benutzen in Python. Es gibt auch die OGR-Bibliothek die verschiedene Formate erstellen kann (z.B. Shape-Files), aber sie ist wesentlich umständlicher zu benützen.
- Sie dürfen Fehler machen (und Sie werden viele Fehler machen). Fehler kann man korrigieren, und man kann daraus lernen.

5.3 Code it!

Versuchen Sie das oben skizzierte Programm fertig zu programmieren. Hier der Code um die Koordinaten anhand einer Adresse zu ermitteln:

```
1    import json, urllib
2    adresse = "Beverinstrasse 2, 7430 Thusis"
3    query_string = adresse.replace(' ', '+')
4    url = 'http://nominatim.openstreetmap.org/search?q=%s&format=
        json' % query_string
5    result = urllib.urlopen(url).read()
6    result_liste = json.loads(result)
7    lat = float(result_liste[0]['lat'])
8    lon = float(result_liste[0]['lon'])
```

Achtung! Eine Adresse kann potentiell 0, 1, oder mehrere Koordinaten enthalten. Der obenstehende Code muss also noch verbessert werden!

Kapitel 6

Objekt-orientiertes Programmieren

Objekt-orientiertes Programmieren (OOP) erleichtert es erheblich, gut strukturierte und einfacher verständliche Programme zu schreiben. Dabei gehen wir hier nicht auf alle Konzepte des OOP ein, sondern wir schauen bloss ein paar Grundkonzepte an.

Objekte erlauben es Variablen und Funktionen die logischerweise zusammengehören in eine eigene Struktur zu packen. Ein Objekt enthält also Variablen und Funktionen, die in diesem Zusammenhang **Eigenschaften** (properties) bzw. **Methoden** (methods) genannt werden. Die Eigenschaften beschreiben den Zustand eines Objektes, während die Methoden Aktionen ausführen, die den Zustand des Objektes potentiell verändern können. Schauen wir doch einfach wie ein solches Objekt definiert wird.

```
1    class MeineKlasse:
2        """Eine Klasse die zeigt wie man Objekte erstellt."""
3        i = 1234
4        a = 'abcd'
5        def f(self):
6            return 'Hallo ' + self.a
```

Hier handelt es sich um die Definition einer **Klasse** (class). In Python werden Klassennamen per Konvention gross geschrieben, um sie von Variablennamen zu unterscheiden. Eine Klasse definiert welche Eigenschaften sich in einem Objekt befinden, und sie definiert die Methoden. Um ein Objekt dieser Klasse zu erhalten, muss man eine Instanz (instance) der Klasse erstellen.

Das tönt esoterisch, ist aber ganz einfach:

```
obj = MeineKlasse()
```

Eine Klasse ist eine Art Schablone die dazu dient neue 'Kopien' zu erstellen. Diese 'Kopien' nennt man dann Instanzen. Ein Objekt ist also eine Instanz einer Klasse. Wir können dies auch mit realen Objekten vergleichen. Wir können zum Beispiel eine Klasse *'Auto'* definieren:

```
class Auto:
    farbe = 'rot'
    marke = 'Audi'
    geschwindigkeit = 0

    def beschleunigen(self, wie_viel):
        self.geschwindigkeit += wie_viel

    def bremsen(self, wie_viel):
        self.geschwindigkeit -= wie_viel
```

Diese Klasse definiert also die Eigenschaften und den Zustand eines Autos, sowie die Möglichkeiten (beschleunigen und bremsen). Aber in diese Definition kann ich mich noch nicht hineinsetzen und meine Kinder zur Schule fahren! Dafür muss ich den Fabrikant beauftragen, eine Instanz dieser Klasse zu erstellen und mir das resultierende Objekt zu verkaufen:

```
gr103826 = Auto()
```

Nun kann ich die Kinder zur Schule fahren (sofern ich auch Kinder habe...):

```
gr103826.beschleunigen(30)  # wohne in einer 30er-Zone...
gr103826.beschleunigen(20)  # und dann in einer 50er-Zone...
gr103826.bremsen(50)    # so einfach ist es normalerweise
    nicht...
gr103826.marke = 'Fiat' # geht normalerweise auch nicht...
```

Die Methoden der Klasse kann ich mit der obigen Punkt-Syntax aufrufen, wie auch die Eigenschaften. Wir haben diese Schreibweise schon häufig gesehen:

```
a = 'hallo'
a.upper()
```

`a` ist eben auch ein Objekt! Und zwar ist es eine Instanz der Klasse `str`, also ein String. Die Erstellung der Instanz ist zwar etwas anders; dieses Verhalten ist jedoch einfach in Python verankert um die Programmierung zu vereinfachen. Oder:

```
import math
math.log(8)
```

Hier ist `math` ein Modul, das eben auch ein Objekt ist... Grundsätzlich ist alles in Python ein Objekt.

6.1 Definition einer Methode

Die Definition einer Methode ist genau gleich wie jene einer einfachen Funktion, mit der Ausnahme dass das erste Argument immer das Objekt selbst ist. Per Konvention wird dieses Argument immer **self** gennant. Um auf andere Methoden oder Eigenschaften des Objektes zugreifen zu können, muss dann `self.methode()` oder `self.eigenschaft` geschrieben werden.

6.2 Konstruktoren und Destruktoren

In einer Klassendefinition ist es möglich, spezielle Methoden zu definieren, die ausgeführt werden, wenn eine Instanz einer Klasse erstellt wird (**Konstruktor**), oder wenn die Instanz zerstört wird (**Destruktor**). Der Konstruktor heisst in Python `__init__` (also doppelter Underscore), und der Destruktor `__del__`. Der Destruktor akzeptiert als einziges Argument nur `self`, während der Konstruktor beliebige Argumente akzeptiert. Hier ein Beispiel:

```
class Auto:
    def __init__(self, marke, farbe='rot'):
        self.marke = marke
        self.farbe = farbe

renault = Auto('Renault')
```

```
7     gelber_audi = Auto('Audi', 'gelb')
```

Der Destruktor wird dazu benützt Dateien zu schliessen, Netzwerkverbindungen zu stoppen, Speicher freizugeben, temporäre Dateien zu löschen usw. Ein Objekt kann explizit gelöscht werden:

```
1     del gelber_audi
```

Falls ein Objekt nicht explizit gelöscht wird, kümmert sich Python darum, von Zeit zu Zeit nicht mehr benützte Objekte aus dem Speicher zu löschen (durch den 'garbage collector', also die Müllabfuhr).

6.3 Vererbung

Eine Klasse kann auf einer anderen Klasse aufbauen. Dabei übernimmt die 'Subklasse' (subclass) alle Eigenschaften und Methoden, fügt neue Eigenschaften und Methoden hinzu, und verändert allenfalls Methoden (*override*). Diesen Mechanismus nennt man *Vererbung* oder *Inheritance*. Dies geht so:

```
1     class Auto:
2         def __init__(self, marke, farbe='rot'):
3             self.marke = marke
4             self.farbe = farbe
5
6     class Lastwagen(Auto):
7         def __init__(self, marke, farbe='rot', gewicht,
          anzahl_achsen):
8             super(Lastwagen, self).__init__(marke, farbe)
9             self.gewicht = gewicht
10            self.anzahl_achsen = anzahl_achsen
```

Auf Linie 6 wird die Subklasse von Auto definiert. Die Logik ist dabei, dass ein Lastwagen ein Spezialfall eines Autos ist. Dafür werden Subklassen typischerweise eingesetzt. Auf Linie 7 wird dann der neue Konstruktor der Lastwagen-Klasse definiert, der nicht nur die Argumente eines Autos hat, sondern auch Lastwagen-spezifische Argumente. Auf Linie 8 wird dann der Konstruktor der Auto-Klasse aufgerufen. Dabei muss man wissen, dass die Auto-Klasse für die Lastwagen-Klasse *Superklasse* genannt

wird. `super(Lastwagen, self)` gibt uns also die Superklasse (also die Auto-Klasse), und wir können den Konstruktor dann manuell aufrufen.

6.4 Code it!

Nun gilt es langsam ernst. Wir haben nun schon viele Konzepte gesehen und brauchen vor allem noch eines: Übung! Hier also eine Programmidee.

Ich muss immer wieder Sprachen lernen und Wörter büffeln. Dafür benutze ich eine Lernkartei, wo jedes Kärtchen ein Wort auf Deutsch auf der Vorderseite hat, und in der Fremdsprache auf der Rückseite. Nun möchte ich ein Programm schreiben, das eine Textdatei mit Wörtern einliest (in zwei Kolonnen, eine Deutsch, die andere z.B. Französisch), und das mich die Französisch-Wörter abfragt. Dabei muss ich die Wörter eintippen und erhalte Feedback ob meine Antwort richtig oder falsch war.

Tipp: Erstellen Sie eine Klasse für die Lernkartei, und eine andere Klasse für jedes Kärtchen.

Benützen Sie das Modul-Template das wir im Kapitel 5 erstellt haben.

Um dieses Programm zu erstellen, fehlen noch zwei Konzepte:

- Die Abfragereihenfolge soll zufällig sein. Dafür benötigen wir das Modul `random`. Schauen Sie in der Python-Dokumentation wie dieses Modul benützt werden kann.

- Die Eingabe von User-Input im Terminal geht ganz einfach mit der Funktion `raw_input`.

Erstellen Sie eine einfache Text-Datei mit Wörtern die Sie anschliessend im Programm laden, z.B.:

```
ein Haus; une maison
eine Umfahrungsstrasse; une route de contournement
ein Computer; un ordinateur
meckern; râler
mit jemandem schimpfen; gronder quelqu'un
```

Kapitel 7

Fehler beseitigen in Skripten

Fehler beseitigen in Skripten und anderen Programmen ist eine häufige Tätigkeit beim Programmieren. Zusammen mit den Tests nimmt es einen grossen Teil der Zeit in Anspruch. Es gibt jedoch verschiedene Arten von Fehler die beheben werden müssen, und es gibt verschiedene Ansätze um solche Fehler zu finden. Im Allgemeinen aber gilt, dass Fehler beseitigen nebst guten Tools vor allem Zeit und Erfahrung braucht. Generell können wir folgende Fehlerarten unterscheiden:

- **Syntaxfehler**. Diese Art Fehler passiert, wenn wir die Syntax-Regeln der Programmiersprache nicht respektieren. In Python kann dies zum Beispiel passieren, wenn der Einschub am Zeilenanfang nicht gleichmässig ist, oder wenn wir ein Strichpunkt anstatt ein Komma benützen um die Argumente einer Funktion zu trennen. Hier ein paar Beispiele:

```
1    def meine-funktion (a=123, b):
2        if (a => 100) then:
3            print "a ist groesser oder gleich 100'
4            print "b ist gleich", b
5          return (a - b / a b
6
7    print meine-funktion(200, 400)
```

  Versuche die verschiedenen Fehler ausfindig zu machen! Es hat mindestens 5 Syntax-Fehler in diesem Beispiel-Code!

  Der Python-Interpreter wird in der Regel den 1. Fehler im Programm ausgeben (eine sogenannte *'Exception'*, mehr dazu weiter unter), und

versuchen den Fehler zu beschreiben und mitteilen, wo im Programm der Fehler passiert ist. Dabei gilt: immer nur den 1. Fehler korrigieren, nachfolgende Fehler können ja einfach durch den 1. Fehler verursacht sein.

- **Semantikfehler**. Diese Art Fehler passiert, wenn wir versuchen eine Funktion inkorrekt zu verwenden. In Python kann dies zum Beispiel passieren, wenn wir versuchen Rechenoperationen mit einer Zeichenkette durchzuführen, oder wenn wir versuchen eine Methode eines Objektes aufzurufen die gar nicht existiert. Das folgende Code-Beispiel versucht eine Liste an Integer-Zahlen zu sortieren und dann alle Elemente in eine Zeichenkette umzwandeln. Python gibt jedoch einen Fehler aus. Was ist daran falsch?

```
1    zahlen = [4, 7, 1, -3, 9, 5]
2    geordnete_zahlen = zahlen.sort()
3    str_zahlen = map(str, geordnete_zahlen)
4    print ', '.join(str_zahlen)
```

  Diese Art Fehler ist etwas schwieriger zu finden als ein Syntax-Fehler. Das Programm läuft zum Teil sogar bis ans Ende, und zum Teil gibt Python einen Fehler aus, der aber häufig bloss die Folge des Semantikfehlers ist, wie im obigen Beispiel. Abhilfe schafft dann häufig das Durchspielen des Codes Schritt für Schritt, und schauen ob alle involvierten Variablen auch den erwarteten Wert haben. Auch muss die Dokumentation sorgfältig angeschaut werden; vielleicht haben wir ja einfach die Art und Weise wie eine Funktion den Input verlangt oder den Output zurückgibt falsch verstanden...

- **Logik-Fehler**. Diese Fehlerart ist häufig anzutreffen, wenn wir die Spezifikation für eine Funktion nicht respektieren. Dies kann zum Beispiel der Fall sein, wenn wir einen Überlegungsfehler machen, aber auch wenn wir eine Zeile Code vergessen, oder eine falsche Funktion oder einen falschen Operator verwenden. Diese Art Fehler muss durch intensive Tests gefunden werden. Das heisst wir müssen jeden Programmbestandteil auf seine Korrektheit testen. Also jede geschriebene Funktion die nicht absolut trivial ist muss analysiert werden ob sie in jedem Fall (auch bei fehlerhaftem Input) das richtige Resultat liefert, und allenfalls eine intelligente Fehlermeldung ausgibt. Dies bedingt aber auch,

dass wir keine 'Monster-Funktionen' schreiben sollten, die dann fast unmöglich zu testen sind.

Ein einfaches Grundrezept zum schnellen Auffinden der Fehler in einem Programm gibt es nicht. Das Beseitigen von Fehlern geschieht mit Hilfe von verschiedenen Tools und Techniken. Grundsätzlich gilt aber immer, den Ort im Programm zu identifizieren von wo an der *'Zustand'* des Programms nicht mehr den Erwartungen entspricht. Das ist der Ort, wo etwas schief läuft. Das bedingt, dass man oben im Programm anfängt zu suchen, oder dass man einen Ort im Programm zu suchen versucht, wo alles noch in Ordnung ist. Dabei muss man sicher stellen, dass alle Bestandteile eines Programmes die erwartete Operation korrekt durchführen. Man kann auch durchaus das Programm so weit vereinfachen bis der verbleibende Teil korrekt funktioniert, und dann Schritt für Schritt die zusätzlichen Funktionen hinzufügen bis das Programm eben nicht mehr korrekt funktioniert.

7.1 Ausnahmen und Versuche

Wenn Python einen Fehler findet, wird das Programm angehalten und eine Fehlermeldung ausgegeben. Das kann dann etwa so aussehen:

```
1 ---------------------------------------------------------------------------
2 TypeError                                 Traceback (most recent call last)
3 /home/user/<ipython-input-6-e99fb8512b48> in <module>()
4 ----> 1 str_zahlen = map(str, geordnete_zahlen)
5
6 TypeError: argument 2 to map() must support iteration
```

Dies ist eine sehr einfache Fehlermeldung. Solche Art Fehlermeldungen werden in Python **'Exception'** (also Ausnahme) genannt. Eine Exception hat immer einen Typ, im obigen Fall ist dies `TypeError`. Das heisst, dass der Variablentyp nicht den Erwartungen entspricht, und zwar auf der Linie 1 des Skripts (Zeile 4 im obigen Output), im Statement *str_zahlen = map(...)*. Auf Linie 6 im obigen Output erfährt man auch, dass das Problem mit dem 2. Argument der *map*-Funktion zusammenhängt. Also kann man in diesem Beispiel mal scahuen was den die Variable *geordnete_zahlen* enthält.

Exeptions können in Python 'aufgefangen' werden, das heisst im Falle einer Exception stoppt das Programm nicht einfach, sondern führt einen ganz bestimmten Code aus. Schauen wir und das folgende Beispiel an:

```
while True:
    try:
        x = int(raw_input('Geben Sie eine Zahl ein: '))
        break
    except ValueError:
        print 'Das war keine gueltige Zahl. Versuchen Sie es
            nochmals.'
```

In diesem Code versuchen wir bewusst einen eventuellen `ValueError` aufzufangen und dem User eine zweite Chance zu geben. Der Code des `try`-Statements wird dabei ausgeführt solange kein Fehler auftritt. Im Falle eines Fehlers springt das Programm zum entsprechenden `except`-Statement, falls vorhanden (andernfalls gibt es entweder ein `try`-Statement weiter oben im Code, oder sonst wird das Programm gestoppt und der Fehler ausgegeben). Somit wird der Code im `except`-Statement nur im Falle eines Fehlers ausgeführt. Es können auch verschiedene `except`-Statements für unterschiedliche Fehlerarten aufeinanderfolgen. Die Fehlerart kann auch weggelassen werden und dann werden alle Fehler aufgefangen, was jedoch nicht empfehlenswert ist da dadurch auch 'ungewollte' Fehler verschleiert werden.

`try/except`-Statements werden vor allem für grössere Programme gebraucht, wo man sicher stellen muss, dass das Programm nicht plötzlich aufhört zu laufen. In diesem Fall muss man jedoch auch sicher stellen, dass der User allenfalls korrekt informiert wird, dass etwas nicht so gelaufen ist wie gewollt. Für kleinere Skripts für den Eigengebrauch die direkt vom Terminal aus laufen lohnt es sich meistens nicht solche `try`-Statements einzubauen, da eine ausführliche Fehlermeldung dann häufig bessere Informationen liefert was falsch gelaufen ist.

Es ist auch möglich selbst eine Exception zu produzieren und so allenfalls einen Programmstopp zu veranlassen. Dies geht so:

```
    raise Exception('Beschrieb was schief gelaufen ist.')
```

Dabei ist es jedoch meist besser eine personalisierte Exception einzusetzen, die dann informativer ist, und die man auch spezifisch auffangen kann mit einem `except`-Statement. Dies geht so:

```
    class Fahrfehler(Exception):
        def __init__(self, value):
            self.value = value
```

```
4           def __str__(self):
5               return repr(self.value)
6
7       if stoppsignal == True and geschwindigkeit > 0:
8           raise Fahrfehler('Am Stopp nicht angehalten!')
```

Intelligent eingesetzte `try`-Statements zusammen mit spezifischen Exceptions können ein Programm stabiler und Fehlermeldungen informativer machen. Jedoch sind `try`-Statements nicht da um jegliche Art von Fehlern aufzufangen und zu ignorieren. Dann ist ein Programm sozusagen *'out of control'* und die Fehler sind viel schwieriger zu finden.

7.2 IDE: Integrated Development Environment

Eine IDE ist eine integrierte Programmierumgebung. Das Hauptziel einer IDE ist es dem Programmierer bei seiner Arbeit zu helfen und all die nötigen Entwicklungstools vom Texteditor bis zum Debugger der hilft Fehler zu finden in einer Umgebung zu vereinen. Zum Teil sind IDEs spezialisiert auf eine Programmiersprache, und zum Teil sind sie sehr generisch. Eine gute IDE kann dementsprechend auch hilfreich sein bei der Suche nach Fehlern.

Hier eine kurze Liste von ein paar IDEs die für Python geeignet sind:

- **IDLE**. Dies ist eine einfache IDE die auf Windows gleich mit Python mitgeliefert wird. Es erlaubt uns Code in mehreren Fenstern zu editieren, hat einen interaktiven Python-Interpreter, und erlaubt auch einfache Debug-Tasks wie das schrittweise Ausführen des Programms durchzuführen.

- **Eclipse**. Eine umfangreiche und flexible IDE für verschiedene Programmiersprachen wie Java, C++, PHP und viele andere. Eclipse hat ein Plug-In-System das es erlaubt weitere Funktionalität hinzuzufügen. Zusammen mit dem Eclipse-Plug-In *PyDev* kann Eclipse auch für Python verwendet werden. Umfangreiche Funktionen wie Code-Komplettierung, Debugger, Interaktive Konsole, Code-Analyse und vieles mehr kann mit Eclipse/PyDev durchgeführt werden.

- **Eric**. Umfangreiche und trotzdem ziemlich intuitiv zu bedienende IDE spezifisch für Python und Ruby. Eric4 wird gebraucht für Python 2.x,

und Eric5 für Python 3.x. Eric ist open-souce und erhältlich auf der Homepage eric-ide.python-projects.org. Eric ist integriert mit *Qt*, einem umfassenden Open-Source Framework für graphische User-Interfaces.

Selbstverständlich gibt es noch viele weitere IDEs. Für jede IDE gibt es Tutorials, und jeder wird seine persönlichen Vorlieben haben. Die Wahl der IDE hängt natürlich auch von unserem Programmier-Projekt ab. Für einfache Skripts mag ein einfacher Text-Editor oder eine IDE wie IDLE genügen. Für komplexere Projekte zum Beispiel ein Programm mit GUI ist dann aber eine umfangreiche IDE eine grosse Hilfe. Dabei ist jede IDE ein wenig unterschiedlich in der Bedienung, auch wenn in den grossen Zügen die Funktionen weitgehend identisch sind.

7.3 print, print, print!

Eine rudimentäre Technik, die ab und zu für das Auffinden von Fehlern gebraucht wird, ist das Ausgeben von spezifischen Variablen an ausgewählten Orten im Programm. So kann der Zustand des Programms teilweise erfasst werden. In Python müsste man dies mit Hilfe von `print`-Statements machen. In diesem Fall kann man mit einem einfachen Text-Editor, einem Python-Interpreter und dem Browser um die Dokumentation zu lesen schon einfache Programme schreiben.

7.4 Python Debugger

Es gibt auch das Python-Modul `pdb` das ein interaktiver Python-Debugger ist, und der auch nützliche Funktionen enthält um in einem Python-Programm Fehler zu finden. `pdb` enhält keinen Text-Editor, und hat auch keine graphische Benutzeroberfläche. Die Interaktion mit dem Debugger geschieht mittels Terminal-Befehlen. `pdb` ist nützlich vor allem für jene die einen Debugger wie `gdb` (*GNU Project Debugger*) kennen.