Einführung in Python (Teil 2)

A very high level language at work

Gregor Lingl

0. Einleitung

In dieser zweiten Folge der Einführung in Python möchte ich *„Python at work“* in einem doppelten Sinn vorführen. An Hand eines Beispielproblems, das ich auch mit meinen Schülern im Wahlpflichtfach Informatik (8. Klasse) behandelt habe, möchte ich

- einerseits zeigen, wie der interaktive Python-Interpreter genutzt werden kann, um sich über Spracheigenschaften von Python Klarheit zu verschaffen und Programmierideen experimentell auszuprobieren, und
- andererseits zeigen, wie man die sehr leistungsfähigen Datentypen von Python dazu nützen kann, relativ komplexe Problemstellungen mit sehr kompaktem und doch effizientem Code zu lösen. Dies wird Ihnen vielleicht eine Idee davon vermitteln, was mit der Aussage gemeint ist, dass Python eine *„very high level language“* ist.
- drittens werde ich dabei – im Anschluss an den Artikel aus PCNEWS-84 - den Datentyp `dictionary` und den Umgang mit Dateien in Python behandeln.

Das Beispiel ist dementsprechend nicht so elementar, dass es schon in den Einführungsunterricht ins Programmieren passt.

- **Anmerkung 1**: Haben Sie Python bei der Hand? Dann arbeiten Sie diesen Artikel mit dem interaktiven Python-Interpreter durch.
- **Anmerkung 2**: Ich verwende Python in der Version 2.3. (Download: http://www.python.org). Da Python doch in relativ starker Entwicklung (unter reger Beteiligung der Benutzer-Gemeinde) ist, finden sich auch in diesem Beispiel einzelne Dinge, die unter Python 2.2 noch ein wenig anders aussahen.

1. Die Problemstellung: Anagramme

Das Programm `anagramm.py`, das hier entwickelt wird, soll aus einer 400 kB großen Textdatei `wordlist.txt`, die mehr als 45000 Wörter enthält, alle Gruppen von Wörtern heraussuchen, die durch Umstellung von Buchstaben auseinander hervorgehen. (Sie können `wordlist.txt` und `minilist.txt` von der Website http://python4kids.net herunterladen).

Sehen wir uns das an einem kleinen Beispiel an. Für die Programmentwicklung und als Beispiel verwenden wir hier eine kleine Datei `minilist.txt`, die folgende Liste von 20 Wörtern enthält, eines pro Zeile:

```
arts
chase
cheap
cheat
drapes
drop
enemy
generate
insect
parsed
rasped
rats
spared
spread
star
sunlight
taint
teach
teenager
Yemen
```

Unser Anagramm-Programm soll daraus folgende Ausgabe erzeugen:

```
Berechnungszeit: 0.0011 s.
Es gibt 5 Anagrammgruppen mit insgesamt 14 Wörtern:
cheat       teach
arts        rats        star
drapes      parsed      rasped      spared      spread
generate    teenager
enemy       Yemen
```

Dies zeigt, was mit „Anagramm“ gemeint ist: Ein Wort ist Anagramm eines anderen Wortes, wenn es durch bloße Umstellung von Buchstaben aus diesem hervor geht, wobei es nicht auf die Groß/Kleinschreibung ankommt. (vgl. `enemy Yemen`).

Da das Programm schließlich mit großen Textdateien operieren soll, sollten wir darauf achten, möglichst effiziente Verfahren einzusetzen.

2. Die Hauptideen der Lösung

1. Wir stellen fest, dass alle Anagramme einer Gruppe verschiedene Anordnungen der selben alphabetisch geordneten Buchstabengruppe sind. Beispiel: `arts`, `rats` und `star` sind Anordnungen der sortierten Buchstabengruppe `arst`. Bezeichnen wir die so gebildete Buchstabengruppe als die Anagrammsignatur eines Wortes, so heißt das, dass `arts`, `rats` und `star` die selbe Anagrammsignatur haben.
2. Wenn wir zu jedem Wort die Anagrammsignatur ausrechnen können, dann können wir jeder Anagrammsignatur eines Wortes, das in der Datei vorkommt, die Liste von Wörtern zuordnen, die diese Signatur haben. Das sieht dann so aus:

```
     cehp : ['cheap']
    acehs : ['chase']
    aceht : ['cheat', 'teach']
   ceinst : ['insect']
 ghilnstu : ['sunlight']
    aintt : ['taint']
     arst : ['arts', 'rats', 'star']
   adeprs : ['drapes', 'parsed', 'rasped', 'spared', 'spread']
 aeeegnrt : ['generate', 'teenager']
    eemny : ['enemy', 'Yemen']
     dopr : ['drop']
```

   Die gesuchten Anagrammgruppen sind nun alle Listen aus dieser Aufstellung, die mehr als ein Element haben.
3. Eine kleinere Teilaufgabe wird dann noch sein, die Wörter aus der Textdatei einzulesen und in eine Wortliste zu stecken.

Sehen wir uns nun an, welche Sprachmittel Python uns zur Verfügung stellt, um diese Aufgabe zu lösen:

3. Programmierung einer Funktion, die aus einem wort dessen Anagramm-Signatur berechnet

Um die gestellte Aufgabe zu lösen, müssen wir mit Strings und Listen arbeiten. Dies sind, wie (fast) alles in Python, Objekte. Objekte verfügen über Methoden. Methoden sind Funktionen, die mit diesen Objekten operieren oder – wie man auch sagt – an diese Objekte gebunden sind. Methoden werden für ein Objekt mit der Syntax `objekt.methode(argumente)` aufgerufen, wie es auch in anderen Sprachen für die Programmierung mit Objekten üblich ist.

Ein paar einfache Beispiele – ohne ausführlichen Kommentar – die wir uns in Pythons Entwicklungsumgebung IDLE mit dem interaktiven Pythoninterpreter ansehen:

Methodenaufrufe für Strings (In der Python-Hilfe zu finden unter: *Python Docs / Library Reference / 2.3.6.1 String methods*)

```
>> "Barbara".count("a")
3
>> s = "Hallo!"
>> s.count("l")
2
>> s.endswith("!")
True
>> s.replace("l","n")
'Hanno!'
```

In Python sind Strings grundsätzlich unveränderliche Objekte. Stringmethoden können daher nicht Strings verändern, sondern immer nur neue Strings (aus den alten) konstruieren und zurückgeben.

Listen sind dagegen veränderliche Objekte und es gibt Methoden, die sie verändern und nichts (d. h. das Objekt `None`) zurückgeben, z. B. die Methode `sort()`. Dagegen ist die Methode `pop()` eine Methode, die die Liste, für die sie aufgerufen wird verändert (indem sie das letzte Element entfernt) und dieses letzte Element zurückgibt:

```
>> [1,0,1,4,2,1,0].count(1)
3
>> mylist = [1,0,1,4,2,1,0]
>> mylist.sort()
>> mylist
[0, 0, 1, 1, 1, 2, 4]
>> mylist.pop()
4
>> mylist
[0, 0, 1, 1, 1, 2]
>>
```

Zur Berechnung der Anagrammsignatur eines Wortes haben wir nun folgendes zu tun:

- einige Methoden von Strings anwenden: `lower()`, `join()`, später `splitlines()`
- die Funktion `list()` benützen, die u. a. aus Strings in Listen von Buchstaben erzeugt
- ausnützen, dass Listen in Python die Methode `sort()` haben, die sie, wie erwähnt, *„in place“* sortiert (also die Listenobjekte selbst verändert.).

Sehen wir uns das Nötige mit dem Pythoninterpreter an:

```
>> wort = "Yemen"
>> wort = wort.lower()
>> wort
'yemen'
>> wort = list(wort)
>> wort
['y', 'e', 'm', 'e', 'n']
>> wort.sort()
>> wort
['e', 'e', 'm', 'n', 'y']
>> "".join(wort) #probiere auch "x".join(wort), "uuu".join(wort)
'eemny'
```

Fein, das ist die Anagrammsignatur von `Yemen`. Somit gelingt es nun leicht eine Funktion zu definieren, die als Argument ein Wort übernimmt und dessen Anagrammsignatur zurückgibt:

```
def anagramm_signatur(wort):
    buchstaben = list(wort.lower())
    buchstaben.sort()
    return "".join(buchstaben)
```

Wer ruft da: „halt - was ist das `join`“? Eine Stringmethode die oben für den Leerstring aufgerufen wurde und als Argument ein Liste von Wörtern übernimmt. Wir sehen uns das einfach an:

```
>> "-".join(["a", "be", "bu", "und raus bist du!"])
'a-be-bu-und raus bist du!'
>> "schluck".join(["a", "b", "c"])
'aschluckbschluckc'
```

Wir schreiben nun den Code für die Funktionsdefinition von `anagramm_signatur` in ein Skript `anagramme.py`. Ausführung dieses Skripts stellt uns dann zunächst nur diese Funktion zur Verfügung.

Um für unser weiteres Vorgehen Wortlisten als Ausgangsmaterial zur Verfügung zu haben beschäftigen wir uns zunächst mit dem Erstellen einer Textdatei aus einer Wortliste.

4. Aus der Textdatei `minilist.txt` eine Liste der Wörter erstellen

Für das Umgehen mit Dateien stellt Python Datei-Objekte zur Verfügung (*Python Docs / Library Reference / 2.3.8 File Objects*). Datei-Objekte, oder `file`-Objekte, werden mit dem eingebauten Konstruktor `file()` erzeugt. Sie haben eine Reihe nützlicher Methoden zum Lesen aus der Datei und zum Schreiben in die Datei. Wir brauchen hier nur die Methode `read()`. Sie liest den gesamten Dateiinhalt als String ein. Aus diesem können wir mittels der oben schon erwähnten String-Methode `splitlines()` die Liste von Zeilen erzeugen. Da unsere Datei aber in jeder Zeile nur ein Wort enthält sind wir damit schon fertig:

```
>> wortdatei = file("minilist.txt")
>> wortdatei
<open file 'minilist.txt', mode 'r' at 0x00A8D460>
```

`mode 'r'` besagt, dass `wortdatei` eine zum Lesen geöffnete Datei ist. (Dies geschieht immer, wenn nichts anderes angegeben ist).

```
>> datei_inhalt = wortdatei.read()
>> datei_inhalt
'arts\nchase\ncheap\ncheat\ndrapes\ndrop\nenemy\ngenerate\ninsect\npa
rsed\nrasped\nrats\nspared\nspread\nstar\nsunlight\ntaint\nteach\ntee
nager\nYemen'
>> wortliste = datei_inhalt.splitlines()
>> wortliste
['arts', 'chase', 'cheap', 'cheat', 'drapes', 'drop', 'enemy',
'generate', 'insect', 'parsed', 'rasped', 'rats', 'spared', 'spread',
'star', 'sunlight', 'taint', 'teach', 'teenager', 'Yemen']
>>
```

Damit haben wir interaktiv eine Liste von Wörtern für Testzwecke erzeugt und schreiben den entsprechenden Code in unser `anagramm`-Skript, gleich unter die Funktionsdefinition von `anagramm_signatur`:

```
wortliste = file("minilist.txt").read().splitlines()
```

Dabei haben wir fortgesetzte Punkt-Notation benützt `file("minilist.txt")` erzeugt ein Dateiobjekt. Dieses hat die Methode `read()`. `file("minilist.txt").read()` erzeugt ein String-Objekt. Dieses hat die Methode `splitlines()`. Letztere, aufgerufen für das Objekt `file("minilist.txt").read()` gibt eine Liste zurück.

5. Programmierung einer Funktion `finde_anagramme`, die aus der Wortliste eine Liste der Anagrammgruppen berechnet und zurückgibt.

Wie in der Einführung schon dargelegt, müssen wir jetzt eine Zuordnung herstellen, die jeder Anagrammsignatur, die zu einem Worte aus der Wortliste gehört, eine Liste aller Wörter zuordnet, die diese Signatur haben.

Für derartige Zuordnungen (engl.: *mapping*) stellt Python den Datentyp `dictionary` zur Verfügung. Jedem Element einer Menge von so genannten Schlüsseln (*keys*) wird ein Wert (*value*) zugeordnet.

dictionaries bestehen daher aus einer Kollektion von Paaren von Objekten, so genannten Schlüssel-Wert-Paaren. Dabei ist zu beachten, dass die Schlüssel unveränderliche Objekte sein müssen. (Es kommen Zahlen, Strings, Tupel in Frage). Die Werte können dagegen beliebige Objekte sein, also auch veränderliche wie Listen.

Wir sehen uns das mit dem interaktiven Python-Interpreter an:

```
>> beruf = { "Jurgen":"Computer-Guru", "Petra":"Sangerin",
"Fritz":"Lehrer", "Klara":"Model"}
```

Die Syntax für die Eingabe von *dictionaries* sieht so aus:. Die Schlüssel-Wert-Paare haben zwischen sich einen Doppelpunkt. Paare werden durch Kommas voneinander getrennt. Das ganze *dictionary* wird in geschwungene Klammern eingeschlossen. Unseres hat den Namen `beruf:`

```
>> beruf
{'Fritz': 'Lehrer', 'Klara': 'Model', 'Jurgen': 'Computer-Guru',
'Petra': 'Sangerin'}
```

Beachten sie, dass die Einträge im Wörterbuch in anderer Reihenfolge auftauchen. Wörterbücher sind ungeordnet. Man hat keinen Einfluss auf irgendeine Art von Reihenfolge der Einträge!

Wir fragen nach den Werten, die zu einzelnen Schlüsseln gehören, in ähnlicher Weise wie nach Elementen von Listen, die zu bestimmten Indizes gehören:

```
>> beruf["Fritz"]
'Lehrer'
>> beruf["Petra"]
'Sangerin'
```

Wir fügen einen Eintrag hinzu:

```
>> beruf["Harry"] = "Programmierer"
>> beruf
{'Fritz': 'Lehrer', 'Harry': 'Programmierer', 'Klara': 'Model',
'Jurgen': 'Computer-Guru', 'Petra': 'Sangerin'}
```

Anmerkung: Einträge können nur zu schon bestehenden *dictionaries* hinzugefügt werden. Will man mit Anweisungen dieser Art ein *dictionary* zusammenstellen, muss man vorher ein leeres *dictionary* erstellen: `meinDict = {}`

Mit derselben Syntax werden auch Einträge in *dictionaries* geändert:

```
>> beruf["Klara"] = "Filmschauspielerin"
>> beruf
{'Fritz': 'Lehrer', 'Harry': 'Programmierer', 'Klara':
'Filmschauspielerin', 'Jurgen': 'Computer-Guru', 'Petra': 'Sangerin'}
```

Der Datentyp Dictionary hat die beiden Methoden `keys()` und `values()`. Sie geben die Schlüssel beziehungsweise die Werte als Liste zurück:

```
>> beruf.keys()
['Fritz', 'Harry', 'Klara', 'Jurgen', 'Petra']
>> beruf.values()
['Lehrer', 'Programmierer', 'Filmschauspielerin', 'Computer-Guru',
'Sangerin']
```

Wie bei jeder Liste kann man abfragen, ob ein Objekt zur Liste gehört:

```
>> "Petra" in beruf.keys()
True
>> "Skater" in beruf.values()
False
(Leider kein Beruf!)
```

Für *dictionaries* hat aber auch der Operator `in` eine Bedeutung: Mit ihm prüft man kürzer – und vor allem wesentlich effizienter als oben, ob ein Objekt ein Schlüssel in einem *dictionary* ist.

```
>> "Petra" in beruf
True
```

Dies wurde mit Python 2.2 eingeführt. Benutzer älterer Versionen haben dafür folgendes zur Verfügung:

```
>> beruf.has_key("Petra")
True
```

Daher kann man nun alle Elemente eines *dictionarys* mit `for`-Schleifen durchlaufen:

```
>> for name in beruf:
       print "Der Beruf von %s ist %s" % (name, beruf[name])
Der Beruf von Fritz ist Lehrer
Der Beruf von Harry ist Programmierer
Der Beruf von Klara ist Filmschauspielerin
Der Beruf von Jurgen ist Computer-Guru
Der Beruf von Petra ist Sangerin
>> beruf["Inge"]
Traceback (most recent call last):
  File "<pyshell#64>", line 1, in ?
    beruf["Inge"]
KeyError: Inge
>>
```

Diese Fehlermeldung `KeyError` – Schlüsselfehler – tritt auf, wenn man nach einem Wert für einen Schlüssel suchst, den es im Wörterbuch nicht gibt. `"Inge"` kommt als Schlüssel im *dictionary* `beruf` nicht vor!

6. Zurück zur Programmierung der Funktion

`finde_anagramme`

Wir gehen davon aus, dass unsere Wortliste bereits gegeben ist und auch die Funktion `anagramm_signatur` bereits fertig programmiert ist:

```
>> wortliste
['arts', 'chase', 'cheap', 'cheat', 'drapes', 'drop', 'enemy',
'generate', 'insect', 'parsed', 'rasped', 'rats', 'spared', 'spread',
'star', 'sunlight', 'taint', 'teach', 'teenager', 'Yemen']
```

Wie wollen nun ein `dictionary` mit den Anagrammsignaturen als Schlüssel und den Anagrammgruppen als Werten erzeugen. Zunächst – noch im Experimentierstadium - erzeugen wir zuerst ein leeres `dictionary` und weisen dann jeder möglichen Anagrammsignatur (als Schlüssel) zunächst einmal den Wert `[]`, also die leere Liste zu:

```
>> d = {}
>> for wort in wortliste:
     sig = anagramm_signatur(wort)
     d[sig] = []
>> d
{'acehp': [], 'acehs': [], 'aceht': [], 'ceinst': [], 'ghilnstu': [],
'aintt': [], 'arst': [], 'adeprs': [], 'aeeegnrt': [], 'eemny': [],
'dopr': []}
```

Aha, deutlich weniger Signaturen, als Wörter. Wieviel?

```
>> len(d)
11
```

`len(d)` liefert die Anzahl der Schlüssel in `d`, als dasselbe wie `len(d.keys())`.

Nun können wir in diese leeren Listen die zugehörigen Wörter hineinpacken. Dazu verwenden wir die Listen-Methode `append`. Wie die funktioniert? So:

```
>> liste = ["a", "be"]
>> liste.append("bu")
>> liste
['a', 'be', 'bu']
>> liste.append("und raus bist du!")
>> liste
['a', 'be', 'bu', 'und raus bist du!']
>>
```

`append()` verändert – wie `sort()` - auch die Liste (es verlängert sie nämlich!). `append()` ist auch eine Methoden ohne Rückgabewert (oder genauer: mit dem Rückgabewert `None`).

Unser Vorhaben verwirklichen wir nun, indem wir nochmals alle Wörter durchgehen, für jedes nochmals die Anagrammsignatur `sig` berechnen und dasWort an die Liste `d[sig]` anhängen. Erinnerung: vor Beginn der Abarbeitung der `for`-Schleife sind alle `d[sig]` leere Listen. Bei der Schleifenausführung werden alle Wörter an die passenden Listen angehängt:

```
>> for wort in wortliste:
     sig = anagramm_signatur(wort)
     d[sig].append(wort)

>> d
{'acehp': ['cheap'], 'acehs': ['chase'], 'aceht': ['cheat', 'teach'],
'ceinst': ['insect'], 'ghilnstu': ['sunlight'], 'aintt': ['taint'],
'arst': ['arts', 'rats', 'star'], 'adeprs': ['drapes', 'parsed',
'rasped', 'spared', 'spread'], 'aeeegnrt': ['generate', 'teenager'],
'eemny': ['enemy', 'Yemen'], 'dopr': ['drop']}
>>
```

Da ist schon das gewünschte `dictionary`. `d.values()` enthält aber noch Listen, die nur ein Wort enthalten. In der Liste der Anagrammgruppen sollen aber nur jene „Werte“ von `d` vorkommen, die mehr als ein Wort enthalten:

```
>> d.values()
[['cheap'], ['chase'], ['cheat', 'teach'], ['insect'], ['sunlight'],
['taint'], ['arts', 'rats', 'star'], ['drapes', 'parsed', 'rasped',
'spared', 'spread'], ['generate', 'teenager'], ['enemy', 'Yemen'],
['drop']]
>> anagramm_gruppen = []
>> for sig in d:
     if len(d[sig]) > 1:
          anagramm_gruppen.append(d[sig])

>> anagramm_gruppen
[['cheat', 'teach'], ['arts', 'rats', 'star'], ['drapes', 'parsed',
'rasped', 'spared', 'spread'], ['generate', 'teenager'], ['enemy',
'Yemen']]
>> for gruppe in anagramm_gruppen:
     print "\t".join(gruppe)
cheat       teach
arts        rats        star
drapes      parsed      rasped      spared      spread
generate    teenager
enemy       Yemen
```

(`"\t"` ist das Tabulator-Zeichen, wie `"\n"` – der *new-line-character* - ein Sonderzeichen.)

Das hätten wir nun. Doch gibt es hier noch eine kleine Unschönheit, die die Programmlaufzeit unnötig verlängert: die Liste der `woerter` muss zweimal abgearbeitet werden und für jedes Wort muss die Anagrammsignatur zweimal berechnet werden. Doch das kann leicht in einem Schleifendurchlauf erledigt und damit besser gemacht werden:

```
for wort in woerter:
    sig = anagramm_signatur(wort)
    if sig not in d:
        d[sig] = []
    d[sig].append(wort)
```

Die `if`-Anweisung ist hier nötig, da nur an bestehende Listen `d[sig]` mit `append` Elemente angehängt werden können. Wenn also die Anagrammsignatur `sig` zum ersten Mal auftaucht ordnen wir ihr (als Schlüssel) zunächst die leere Liste (als Wert) zu.

7. Ein paar kleine Feinheiten

Wenn man sich ein bisschen in der Python-Dokumentation zum Thema Dictionaries umsieht, (*Python Docs, Python Library Reference, 2.3.7 Mapping Types*) bemerkt man, dass es für solche Fälle wo Werte in einem `dictionary` verändert werden sollen, eine passende Methode gibt, die Schlüsseln default-Werte zuordnet: `setdefault(key, val)`. Diese Methode gibt den zu `key` gehörigen Wert zurück, wenn `key` existiert. Andernfalls setzt sie ihn auf `val` und gibt `val` zurück. `d.setdefault(sig, [])` gibt daher die zu `sig` gehörige Liste zurück, wenn `sig` bereits ein Schlüssel ist und andernfalls setzt es den Wert `[]` und gibt diese leere Liste zurück. Damit kann obige Schleife noch kürzer gefasst werden:

```
for wort in woerter:
    d.setdefault(anagramm_signatur(wort), []).append(wort)
```

Und nun kann wie gehabt aus `d` die Liste der Anagrammgruppen ermittelt werden.

Oder wir wählen eine alternative Syntax, so genannte *list-comprehensions*. Diese lehnen sich etwas an die aus der Mathematik bekannte Mengennotation an. Sie sind nützlich, wenn aus gegebenen Listen (oder anderen Sequenzen) neue erzeugt werden sollen. **Beispiele**:

```
>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
>> [x for x in range(10) if x**2%5==1]
[1, 4, 6, 9]
```

Das können wir auch verwenden um eine Liste von Anagrammgruppen zu erzeugen:

```
[ag for ag in d.values() if len(ag) > 1]
```

Wir entschließen uns nun noch auch die Gesamtzahl der Wörter in den Gruppen von Anagrammen mit der (seit Version 2.3) eingebauten Funktion `sum()` zu berechnen, der wir das Ergebnis einer *list-comprehension* als Agument übergeben, die Funktion `clock()` aus dem Modul `time` zu importieren, damit wir auf der Computeruhr nachsehen können wie spät es ist – und daraus die Laufzeit unseres Programms zu berechnen.

Damit gelangen wir zu folgendem Code:

8. Das fertige Programm `anagramme.py`

```
from time import clock
def anagramm_signatur(wort):
    buchstaben = list(wort.lower())
    buchstaben.sort()
    return ''.join(buchstaben)

def finde_anagramme(woerter):
    d = {}
    for wort in woerter:
        d.setdefault(anagramm_signatur(wort), []).append(wort)
    return [ag for ag in d.values() if len(ag) > 1]

t1 = clock()
woerter = file("wordlist.txt").read().splitlines()
anagruppen = finde_anagramme(woerter)
t2 = clock()
print "Berechnungszeit: %5.2f s." % (t2-t1)
print "Es gibt %d Anagrammgruppen" % len(anagruppen)
print "mit insgesamt %d Wörtern" % sum([len(ag) for ag in anagruppen])
print
raw_input("Eingabe-Taste druecken\n")
for anagramme in anagruppen:
    print '\t'.join(anagramme)
```

Ein Programmlauf liefert mir (auf einem 2.66 GHz- Rechner unter Windows-XP) das folgende Ergebnis:

```
Berechnungszeit:  0.53 s.
Es gibt 2531 Anagrammgruppen
mit insgesamt 5683 Wörtern
Eingabe-Taste druecken
Remus       serum
horse       shore
strain      trains
disowned    downside
bluer       ruble
Akron       Koran
fierce      Recife
Erich       Reich
bluest      bustle      subtle
....
```

9. Kurze Schlussbetrachtung

Sehen wir einmal von der Ausgabe ab, so finden wir, dass knapp mehr als zehn Zeilen Code reichen um die Anagrammgruppen aus `wordlist.txt` zu berechnen. Der Code hat eine klare Struktur und ist meines Erachtens gut lesbar.

Obwohl Python eine interpretierte Sprache ist, hat das Programm ein äußerst praktikables Laufzeitverhalten, das noch dazu proportional zur Anzahl der zu verarbeitenden Wörter in der Textdatei ist. Dies liegt natürlich daran, dass so leistungsfähige Datentypen wie Listen und *dictionaries* in Python eingebaut und hochgradig optimiert sind.

Anregung

Ich wäre sehr interessiert an äquivalenten anagramm-Programmen, die in anderen Programmiersprachen (C (C++), Pascal (Delphi), Java, VisualBasic, PHP, Perl, usw.) implementiert sind (oder vielleicht sogar in dem als Wollmilchsau berühmten Excel + VBA), um zu sehen:

- welche Unterschiede gibt es im Programmieraufwand in verschiedenen Sprachen
- welche Unterschiede gibt es im Laufzeitverhalten

Sollten Sie, geneigter Leser, solche zufällig haben – oder erzeugen wollen, senden Sie mir bitte ein Exemplar davon an *glingl@aon.at* . Wenn Sie Kommentare oder Vorschläge zu der hier gezeigten Problemlösung haben, sind diese natürlich auch sehr willkommen.

Sollten ein paar davon zusammenkommen, könnte vielleicht eine kleine Zusammenschau für dieses Blatt dabei herausschauen.

In dieser Folge haben wir schon stark von den Fähigkeiten von Python-Objekten Gebrauch gemacht, deren Typ in Python eingebaut ist. In der nächsten Folge soll es um objektorientierte Programmierung im engeren Sinn gehen: um die Programmierung benutzerdefinierter Klassen.

Literatur

Zusätzlich zu den in PCNEWS-84 genannten Quellen ist inzwischen erschienen:

Michael Weigend: Python GE-PACKT in der GE-PACKT-Reihe des mitp-Verlags. Ein sehr preisgünstiges und nützliches Nachschlagewerk das aber auch viele kurze und klare Beispiele enthält.