Python2

February 23, 2020

1 Weiterführende Python Konzepte

Eingebaute Funktionen, Bedingte Ausdrücke, List und Dict Comprehension, File IO, Fehlerbehandlung

1.1 Eingebaute Funktionen: Aufzählungen mit Index

Manchmal möchte man durch Listen iterieren, inklusive Index für jedes Element. Dies geht mit `enumerate( ... )`

```
In [1]: data = [ 'Spam', 'Eggs', 'Ham' ]

        for index, value in enumerate(data):
            print(index, ":", value)
```

```
0 : Spam
1 : Eggs
2 : Ham
```

1.2 Eingebaute Funktionen: Listen kombinieren

Es gibt eine einfache Möglichkeit, Listen element-weise zusammenzuführen (wie ein Reissverschluss!): `zip( ... )`

```
In [2]: places = [ 'Zurich', 'Basel', 'Bern']
        plz = [ 8000, 4000, 3000, ]

        # Create a list of tuples of the nth element of each input list
        l = list(zip(places, plz))
        print('l =', l)

        # Create a dictionary from places to plz
        d = dict(zip(places, plz))
        print('d =', d)
```

```
l = [('Zurich', 8000), ('Basel', 4000), ('Bern', 3000)]
d = {'Zurich': 8000, 'Basel': 4000, 'Bern': 3000}
```

1.3 Bedingte Ausdrücke

In Python kann der Wert eines Ausdrucks von einer Bedingung abhaengen (als Teil des Ausdrucks!)

Beispiel: Collaz Folge:

```
In [3]: a = 17
        while a  != 1:
            print(a, end='  ')
            a = a // 2 if a % 2 == 0 else a * 3 +1
        print(1)
```

```
17  52  26  13  40  20  10  5  16  8  4  2  1
```

In folgendem Beispiel kommt dies etwas besser zum Ausdruck:

```
In [4]: def output(n):
            print('I see', n, 'mouse' if n == 1 else 'mice')

        output(1)
        output(2)
```

```
I see 1 mouse
I see 2 mice
```

1.4 List Comprehension

- Python bietet eine sehr angenehme Möglichkeit an, Listen deklarativ zu erstellen
- Ähnliche Denkweise wie bei 'map' und 'filter' bei funktionalen Sprachen

Beispiel: Eine Sequenz von Zahlen in eine Liste von Zahlen parsen.

Die Funktion `.split()` auf einem String zerteilt den String in eine Liste von Strings, wobei standardmässig angenommen wird, dass bei Leerzeichen gesplittet werden soll.

```
In [5]: line = '1 4 28 -17 42'
        s_list = line.split()
        n_list = [ int(x) for x in s_list ]

        print('line = \'', line, '\'', sep='')
        print('s_list =', s_list)
        print('n_list =', n_list)
```

```
line = '1 4 28 -17 42'
s_list = ['1', '4', '28', '-17', '42']
n_list = [1, 4, 28, -17, 42]
```

Dasselbe kombiniert in einem einzigen Ausdruck:

```
In [6]: n_list = [ int(x) for x in '1 4 28 -17 42'.split() ]
        print('n_list =', n_list)
```

```
n_list = [1, 4, 28, -17, 42]
```

Beispiel: Leerschläge vorne und hinten eliminieren.

Die Funktion `.strip()` auf einem String gibt einen neuen String zurück, bei dem vorne und hinten allfällige Leerzeichen entfernt wurden.

Funktion `.join()` auf einem String erstellt eine Zeichenkette, in der die in der als Parameter übergebenen Listenelemente mittels dem

```
In [7]: line = [ '  some eggs   ', '  slice of ham ', ' a lot of spam ' ]
        cleaned = [ item.strip() for item in line ]

        print('line =', ', '.join(line))
        print('cleaned =', ', '.join(cleaned))
```

```
line =   some eggs   ,   slice of ham ,  a lot of spam
cleaned = some eggs, slice of ham, a lot of spam
```

1.5 Dict Comprehension

Wie bei Listen, aber mit Key/Value Paaren

Beispiel: Daten extrahieren aus einem Dict:

```
In [8]: data = {
            'Spam' : { 'Amount' : 12, 'Price': 0.45 },
            'Eggs' : {                'Price': 0.8  },
            'Ham'  : { 'Amount': 5,   'Price': 1.20 }
        }

        # show the record for 'Spam', just to understand what's stored
        data['Spam']
```

```
Out[8]: {'Amount': 12, 'Price': 0.45}
```

Mittels der Funktion `.items()` auf einem Dict kann man durch alle Key/Value Paare des Dicts iterieren.

```
In [9]: data.items()
```

```
Out[9]: dict_items([('Spam', {'Amount': 12, 'Price': 0.45}), ('Eggs', {'Price': 0.8}), ('Ham',
```

Wir können also zum Beispiel alle Keys und Values ausgeben in einer `for`-Loop. Dies ist analog zum iterieren durch eine Liste, nur dass wir hier nun zwei Variablen haben: `key` und `value` (diese können beliebig benannt werden):

```
In [10]: for key, value in data.items():
             print(key, '  ', value)
```

```
Spam    {'Amount': 12, 'Price': 0.45}
Eggs    {'Price': 0.8}
Ham    {'Amount': 5, 'Price': 1.2}
```

Hier nochmals das gleiche, aber mit anderen Iterations-Variablennamen:

```
In [11]: for product, information in data.items():
             print(product, '  ', information)

Spam    {'Amount': 12, 'Price': 0.45}
Eggs    {'Price': 0.8}
Ham    {'Amount': 5, 'Price': 1.2}
```

Wir legen uns gleich eine Funktion bereit, um später einfach ein Dict auszudrucken:

```
In [12]: def printDict(dict):
             for key, value in dict.items():
                 print(key, '  ', value)
```

Jetzt kommt der interessante Moment. Wir können nicht nur Dicts zerlegen und darüber iterieren, wir können genau gleich einfach Dicts erstellen.

Dies nennen wir **Dict-Comprehension**.

Beispiel: Wir wollen die Einzelpreise der Produkte in ein Dict abspeichern. Wir wollen ein neues Dict erstellen, in dem für jedes Produkt direkt der Einzelpreis als Value abgespeichert ist:

```
In [13]: prices = { key : value['Price']
                    for key, value in data.items()}

          printDict(prices)

Spam    0.45
Eggs    0.8
Ham    1.2
```

Beispiel: Wir gehen einen Schritt weiter: Wir wollen durch das `data` Dictionary iterieren und bauen daraus ein anderes Dictionary mit den selben Keys (`key`), und als value multiplizieren wir jeweils das `Amount`-Feld und das `Price`-Feld zusammen.

Interessant: Wir können also direkt die Berechnung `value['Amount'] * value['Price']` innerhalb der Dict-Comprehension ausführen.

1. Versuch: (Nebenbei: Hier sehen wir schon etwas 'Fehlerbehandlung', kommt später)

```
In [14]: try:

             total_prices = { key : value['Amount'] * value['Price']
                 for key, value in data.items()}
```

```
            printDict(total_prices)

        except KeyError as e:
            print('Whoops! Key not found:', e.args[0])
```

```
Whoops! Key not found: Amount
```

Wir haben ein Problem: Im Eintrag von `Eggs` steht kein `Amount`-Feld. Dies führt dazu, dass wir bei der Berechnung auf ein nicht-existierendes Feld zugreifen. Das führt zu einem Fehler bei der Ausführung.

Wir können, wie oben gelernt, direkt in Anweisung Bedingung hinzufügen mit einem nachgestellten `if`. Das können wir nun ausnützten!

Wir stellen zusätzlich die Bedingung, dass das Feld mit Key `Amount` in den Produktdaten stehen muss, ansonsten wird das Produkt nicht in das resultierende Dict aufgenommen:

```
In [15]: total_prices = { key : value['Amount'] * value['Price']
             for key, value in data.items()
             if 'Amount' in value }

         printDict(total_prices)
```

```
Spam    5.4
Ham    6.0
```

2 File IO

Lesen und schreiben von Dateien.

- Files können mit dem Befehl `open` geöffnet werden.
- Damit Files automatisch wieder geschlossen werden, muss das innerhalb eines sogenannten `with` Blocks geschehen.

Beispiel: CSV File schreiben (unser `prices` Dict von oben):

```
In [16]: import csv

         with open('product_prices.csv', 'w', newline='') as csvfile:
             headers = ['Product', 'Price']
             writer = csv.DictWriter(csvfile, fieldnames=headers)
             writer.writeheader()
             for product, price in prices.items():
                 writer.writerow({
                     'Product': product,
                     'Price': price
                 })
```

```
        print('Printing the raw file content to see if it worked:\n')
        with open('product_prices.csv', 'r') as file:
            print(file.read())
```

```
Printing the raw file content to see if it worked:

Product,Price
Spam,0.45
Eggs,0.8
Ham,1.2
```

Beispiel: CSV File als Dicts einlesen:

```
In [17]: import csv

         with open('product_prices.csv', newline='') as csvfile:
             reader = csv.DictReader(csvfile)
             for row in reader:
                 printDict(row)
                 print()
```

```
Product    Spam
Price    0.45

Product    Eggs
Price    0.8

Product    Ham
Price    1.2
```

3 Ausnahmebehandlung

Gegeben folgender Code:

```
x = int(input('A number please: '))
```

Wenn keine Zahl eingegeben wird, stürzt das Programm ab:

```
Traceback (most recent call last):
    File "main.py", line 1, in <module>
        x = int(input('A number please: '))
ValueError: invalid literal for int() with base 10: 'a'
```

Wir können diesen Fehler auffangen und entsprechend reagieren. Ausserdem sehen wir hier noch, wie Funktionen in Python aufgerufen und Rückgabewerte zurückgegeben werden.

```
In [18]: def readNumber(input):
             x = 0
             try:
                 x = int(input)
             except ValueError:
                 print('Oh boy, that was no number...')
             return x


         print('Input: \'Spam\'  Result:', readNumber('Spam'))
         print('Input: \'42\'    Result:', readNumber('42'))
```

```
Oh boy, that was no number...
Input: 'Spam'  Result: 0
Input: '42'    Result: 42
```