

Generators

Was sind eigentlich iterables?

```
4 x = ["a", "b", "c"]
5 y = "def"
6 z = 123
7
8 for var in [x, y, z]:
9     print("variable:", var)
10    for elem in var:
11        print(elem, end=" ")
12    print("\n")
13
```

```
variable: ['a', 'b', 'c']
```

```
a b c
```

```
variable: def
```

```
d e f
```

```
variable: 123
```

```
Traceback (most recent call last):
```

```
File "C:\Users\Esad\Desktop\pi.py", line 10, in <module>
```

```
    for elem in var:
```

```
TypeError: 'int' object is not iterable
```

```
[Finished in 157ms]
```

Was sind eigentlich iterables?

```
4 x = ["a", "b", "c"]
5 y = "def"
6 z = 123
7
8 for var in [x, y, z]:
9     print("variable:", var)
10    if "__iter__" in dir(var):
11        print("=> is iterable!")
12    else:
13        print("=> is not iterable!")
14
```

```
variable: ['a', 'b', 'c']
=> is iterable!
variable: def
=> is iterable!
variable: 123
=> is not iterable!
```

Fazit: Ein Typ ist Iterable, wenn die Methode `__iter__()` existiert (-> 🐦)

for-Schleifen haben gelogen!

- Vor dem Iterieren rufen sie die `iter()`-Funktion auf*:

```
for elem in [1, 2, 3]:  
    print(elem)  
  
# ist eigentlich:  
  
for elem in iter([1, 2, 3]):  
    print(elem)
```

- **nicht immer aber für das Verständnis kann man davon ausgehen*

Die `iter()`-Funktion

```
x = iter([1, 2, 3])  
print(x)          # Output: <dict_keyiterator object at 0x0000025E96216CF0>  
print(dir(x))     # Output: [..., __next__, ...]
```

- Geht mit allen iterables
 - Auch mit Elementen, die die `.__getitem__()`-Methode haben
- Mit der `next()`-Funktion kommt man **nach und nach** an die Elemente:

```
x = iter([1, 2, 3])  
print(next(x))    # Output: 1  
print(next(x))    # Output: 2  
print(next(x))    # Output: 3  
print(next(x))    # Output: StopIteration
```

Lazy Evaluation

- *Faule* Auswertung (eher *bequeme* Auswertung)
- Grundsatz: „Ausdrücke nur dann auswerten, wenn man sie braucht“

```
import sys

# Listen sind nicht lazy
print(sys.getsizeof([0]))      # 64 Bytes
print(sys.getsizeof([0, 1]))  # 72 Bytes

# range objects sind lazy af
print(sys.getsizeof(range(10)))      # 48 Bytes
print(sys.getsizeof(range(100)))     # 48 Bytes
print(sys.getsizeof(range(100000000))) # 48 Bytes
```

- Vorteil: speed
- Nachteil: verwirrend

Generator Functions

- Wiederholung:
(i for i in <iterable>) erzeugt eine Generator Expression
(kein Tupel!)
- Beispiel:

```
x = (i for i in range(3))  
print(x) # <generator object <genexpr> at 0x000001B53DEF9A10>  
print(next(x)) # 0  
print(next(x)) # 1  
print(next(x)) # 2  
print(next(x)) # StopIteration
```

List Comprehension vs. Generator

```
list_compr = [i for i in range(10000)]
gen_expr = (i for i in range(10000))

# Größenunterschiede:
print(sys.getsizeof(list_compr)) # 85176 Bytes :/
print(sys.getsizeof(gen_expr))   # 104 Bytes :)

# Trotzdem kommt man an alle Werte:
print(sum(list_compr)) # 49995000
print(sum(gen_expr))   # 49995000

# Aber nicht alles geht:
print(len(list_compr)) # 10000
print(len(gen_expr))   # TypeError
```


Generator Functions

- Sehen aus wie normale Functions ABER `yield` statt `return`

```
def fun():  
    return 10  
  
def gen():  
    yield 10  
  
print(fun())      # 10  
print(gen())      # <generator object gen at 0x0000026BDD329A10>  
print(next(gen())) # 10
```

Generator Functions

```
def fun(x):  
    return x * 2  
    return x * 3  
    return x * 4
```

```
nr = fun(10)  
print(nr)    # 20  
print(nr)    # 20  
print(nr)    # 20
```

```
def gen(x):  
    yield x * 2  
    yield x * 3  
    yield x * 4
```

```
nr = gen(10)  
print(next(nr)) # 20  
print(next(nr)) # 30  
print(next(nr)) # 40
```

return beendet die Funktion, yield nicht

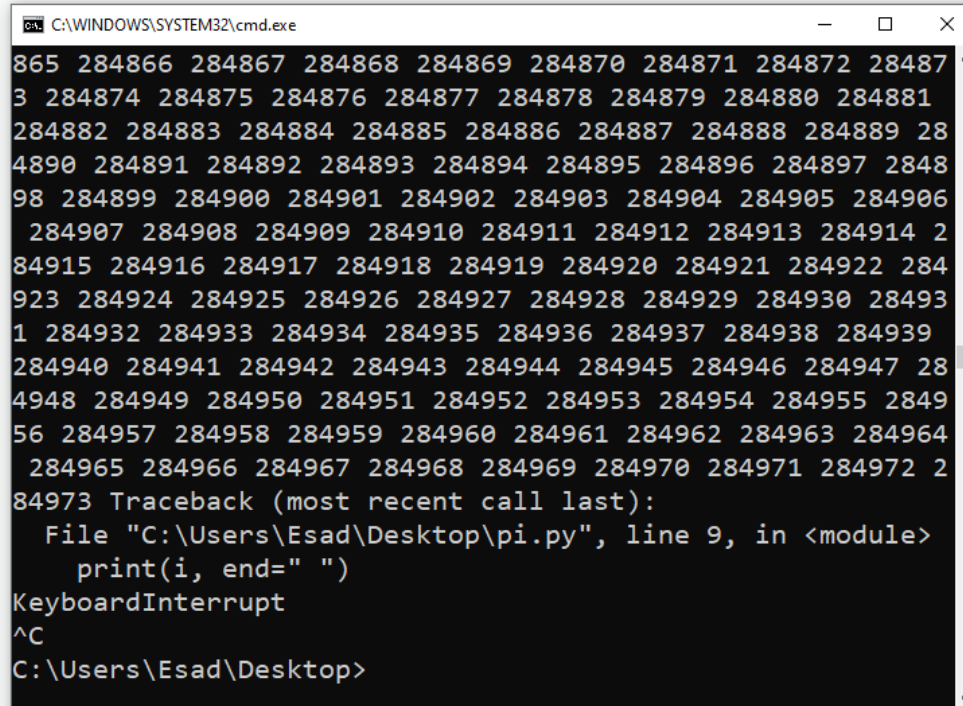
Generator Functions

- Funktionen sind simpel:
 - werde aufgerufen -> gebe etwas zurück -> vergiss alles
- Generator haben einen Zustand (state):
 - werde aufgerufen -> gebe etwas zurück -> mach weiter bis zum nächsten yield -> merke dir den aktuellen Zustand
- Sehr nützlich, wenn die Funktion viel zurückgibt
- Zum Beispiel beim Einlesen von sehr großen Dateien (csv, json etc.)
 - Vorgehen: mit yield Zeile für Zeile lesen
 - So kann theoretisch eine Datei gelesen werden, die größer als der vorhandene Speicher ist

Generator Functions

- Beispiel: unendliche iterables

```
def inf_iterable():  
    x = 0  
    while True:  
        yield x  
        x += 1  
  
for i in inf_iterable():  
    print(i, end=" ")
```



```
C:\WINDOWS\SYSTEM32\cmd.exe  
865 284866 284867 284868 284869 284870 284871 284872 284873 284874 284875 284876 284877 284878 284879 284880 284881  
284882 284883 284884 284885 284886 284887 284888 284889 284890 284891 284892 284893 284894 284895 284896 284897 284898  
284899 284900 284901 284902 284903 284904 284905 284906 284907 284908 284909 284910 284911 284912 284913 284914 284915  
284916 284917 284918 284919 284920 284921 284922 284923 284924 284925 284926 284927 284928 284929 284930 284931  
284932 284933 284934 284935 284936 284937 284938 284939 284940 284941 284942 284943 284944 284945 284946 284947 284948  
284949 284950 284951 284952 284953 284954 284955 284956 284957 284958 284959 284960 284961 284962 284963 284964  
284965 284966 284967 284968 284969 284970 284971 284972 284973  
Traceback (most recent call last):  
  File "C:\Users\Esad\Desktop\pi.py", line 9, in <module>  
    print(i, end=" ")  
KeyboardInterrupt  
^C  
C:\Users\Esad\Desktop>
```

```
print(sys.getsizeof(inf_iterable())) # 104 Bytes
```

Generator Functions

- Beispiel: range mit inklusivem Ende

```
def my_range(start, stop, step=1):  
    """Like range() but stop is inclusive!"""  
    while True:  
        yield start  
        start += step  
        if start > stop:  
            break  
  
print(list(range(1, 3)))      # [1, 2]  
print(list(my_range(1, 3)))  # [1, 2, 3]
```

Further Reading

- Generator Objekte haben 3 Methoden: .send() .throw() .close()
 - <https://realpython.com/introduction-to-python-generators/#using-advanced-generator-methods>
- Python Docs:
<https://docs.python.org/3/library/stdtypes.html#iterator-types>

Aufgabe: Baue eine for-Schleife ohne for

- for-Schleifen nutzen intern die `iter()`- und `next()`-Funktionen
- Baue mithilfe einer `while` Schleife die Funktionsweise einer `for`-Schleife nach:
 - for-Schleifen holen nach und nach Elemente bis es nicht mehr geht
 - Aber was genau heißt das? Wann hört ein Iterator genau auf?

Praxisbeispiel

- Aufgabe 1: Implementieren eines Putzplans mit zufälliger Aufgabenzuordnung
 - Siehe 02_Übung.py
- Aufgabe 2: Aus Dateien nur bestimmte Zeilen speichern
 - Siehe 02_Übung.py

Hausaufgabe

- Siehe 02_Hausi.py