Fortgeschrittenes Python

Lamdba-Ausdrücke, Listcomprehension, Generatoren und Dekorateure

Jan Digeser

Programmierung in Python, 30.05.2018

Inhalt

Vererbung und abstrakte Klassen I

- Subklassen erben Methoden und Variablen
- Methoden können überschrieben werden
- *isinstance(i,A)* ist *True* wenn i Instanz von A oder einer Subklasse von A
- Mehrfachvererbung wird unterstützt

Vererbung und abstrakte Klassen II

Beispiel (Vererbung)

```
1  class Vogel:
2      def fly(self):
3          print("Flap Flap Flap")
4
5  class Geier(Vogel):
6      aasfresser = True
7
8  class Pinguin(Vogel):
9      def fly(self):
10         raise NotImplementedError("Kann nicht fliegen")
```

Vererbung und abstrakte Klassen III

- Abstrakte Klassen erben von *abc.ABC*
- Alle Methoden die mit *@abstractmethod* dekoriert sind müssen implementiert werden

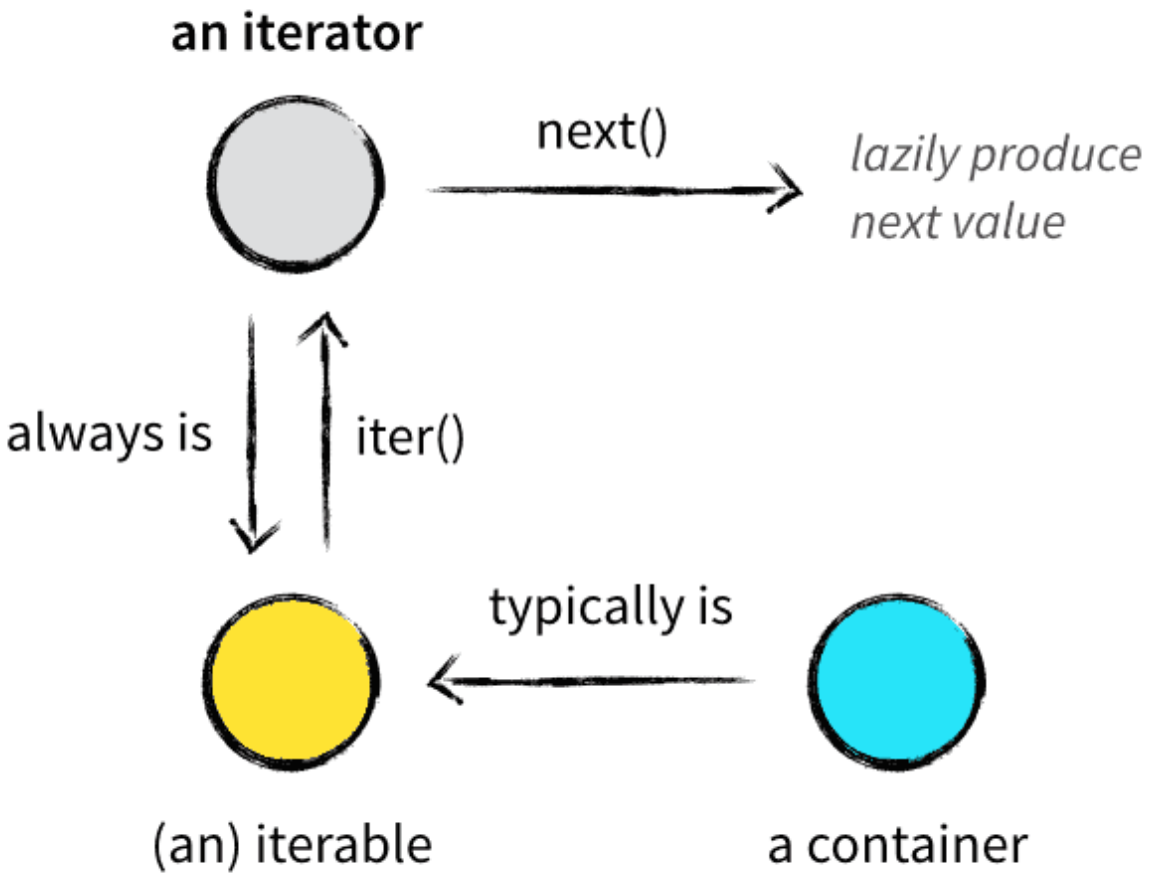
Beispiel (Abstrakte Klassen)

```
1 from abc import ABC, ABCMeta, abstractmethod
2
3 class MyABC(ABC): # class MyABC(metaclass=ABCMeta)
4
5     @abstractmethod
6     def test(self): ...
```

Iteratoren und Iterable I

- **Iteratoren** liefern ein Element, jedes mal wenn man sie danach fragt (mit *next()*)
- **Iterable** sind Objekte, von denen man einen Iterator erstellen kann (mit *iter()*)
- (Fast) alle Container sind *iterable*
- Sehr viele Funktionen benötigen *Iterables* als Eingabe
- Wenn ein Iterator erschöpft ist, raised er *StopIteration* beim Versuch das nächste Element zu holen

Iteratoren und Iterable II



Quelle: `http://nvie.com/img/relationships.png`

Iteratoren und Iterable III

```
1  class Iterable(ABC):
2
3      @abstractmethod
4      def __iter__(self) -> Iterator: ...
5
6
7  class Iterator(ABC):
8
9      def __iter__(self) -> Iterator:
10         return self
11
12     @abstractmethod
13     def __next__(self) -> object: ...
```

Was sind Lambda-Ausdrücke?

- Anonyme (unbenannte) Funktionen
- Stammen aus dem Lambda-Kalkül von Alonzo Church (1936)
- Bilden Grundlage für funktionale Programmierung

Syntax

lambda [parameter] : ausdruck

Gebrauch von Lambda-Ausrücken I

Beispiel (Als normale Funktion)

```
>>> quadriere = lambda x : x * x
>>> quadriere
<function <lambda> at 0x053F3D20>
>>> quadriere(3)
9
```

Map, Filter & Reduce I

- **map(f, iterable, ...)** wendet eine Funktion *f* auf die Elemente von einer oder mehreren *iterable* an
- **filter(f, iterable)** konstruiert einen Iterator mit den Elementen aus *iterable*, für die *f* wahr ist
- **itertools.reduce(f, iterable[, init])** wendet *f* auf die ersten beiden Elemente von *iterable* an und ersetzt diese, bis nur ein Wert übrig bleibt

Map, Filter & Reduce II

Beispiel (map, filter, reduce)

```
>>> list( map( lambda x: x*2, [1, 2, 3, 4, 5, 6]))
[2, 4, 6, 8, 10, 12]
>>> list( filter( lambda x : x%3 == 1, [1, 2, 3, 4, 5, 6]))
[1, 4]
>>> list( reduce( lambda a, b: [*a, a[-1]+b], [10, 5, -3], [0]))
[0, 10, 15, 12]
```

Erzeugen von Containern I

- Map, Filter und Reduce sind nicht ideal um einfache Container zu schaffen
- Reduce wurde mit Python 3 nach functools verbannt
- Comprehensions sind intuitiver, einfacher zu lesen und sehr effizient

Grammatik

comprehension = ausdruck comp_for
comp_for = "for" var "in" smth [comp_iter]
comp_iter = comp_for | comp_if
comp_if = "if" condition [comp_iter]

Erzeugen von Containern II

- Comprehensions sind mit Listen, Mengen und Dictionaries möglich
- Man benutzt dabei eckige Klammern für Listen und geschweifte Klammern für Mengen und Dictionaries

Beispiel (einfache Comprehensions)

```
>>> [ x*x for x in  [1,2,3,4,5]  ]
[1, 4, 9, 16, 25]
>>> { x for x in range(7) if x % 3 == 1 }
{1, 4}
>>> { a: len(a) for a in ['aaa', 'a', 'abcd'] }
{'aaa': 3, 'a': 1, 'abcd': 4}
```

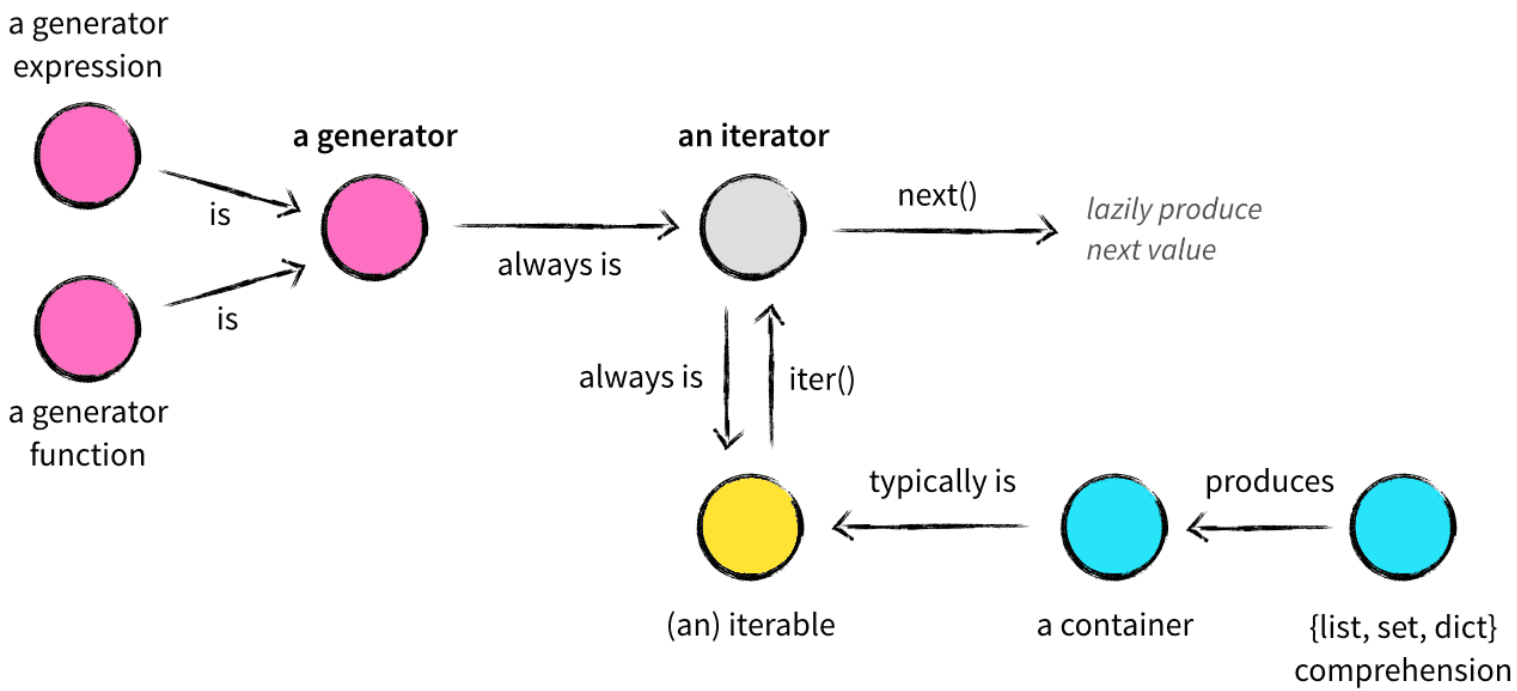
Erzeugen von Containern III

Beispiel

```
1  """
2  Sei array ein 2D-Array zum Beispiel ein Spielfeld
3  Oft benoetigt man die Nachbarn einer Zelle
4  (z.B. Minesweeper, Game of Life)
5  pos_x, pos_y ist die Position dieser Zelle
6  """
7
8  neighbors = [array[pos_x + xoff][pos_y + yoff]
9                  for xoff in {-1,0,1}
10                 for yoff in {-1,0,1}
11                 if not xoff == yoff == 0
12             ]
```

Wie schreibe ich einfacher Iteratoren?

- Warum?
  - Klassen sind nicht schön
  - Man muss eine bestimmte Form befolgen, die Platz verbraucht



Quelle: `http://nvie.com/img/relationships.png`

Generator-Expressions

- Für einfache Iteratoren
- Gut für Iteratoren über vorhandene Daten
- Kann jedoch keine inneren Zustände haben

Grammatik

gen_express = "(" comprehension ")"

Beispiel

```
>>> a = iter( [ x for x in range(10) ] )
>>> b = ( x for x in range(10) )
>>> all( x == y for x, y in zip(a,b) )
True
```

Generator-Funktionen I

- Sehen Funktionen sehr ähnlich
- Benutzen das Keyword *yield*
- Können innere Zustände haben

Syntax

```
def name(*args, **kwargs):
    ...
    yield x
    ...

gen = name(*args, **kwargs)
```

Generator-Funktionen II

Beispiel

```
>>> def fib():
...     old, current = 0, 1
...     while True
...         old, current = current, current + old
...         yield old

>>> f = fib()
>>> [ next(f) for i in range(10) ]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Generator-Funktionen III

- Der Generator arbeitet erst wenn *next()* aufgerufen wird
- Es wird bis zum nächsten *yield* ausgeführt, dann mit aktuellen Zustand pausiert
- Da immer nur ein Element berechnet und ausgegeben wird, sehr effizient
- Zusätzliche Methoden
  - **send(value)** aktuelles *yield* nimmt *value* an
  - **throw(exception)** wirft *exception* an der Stelle von *yield*
  - **close()** schließt Generator (wirft *GeneratorExit*)
- Mit *return* kann der Generator immer beendet werden

Generator-Funktionen IV

- Generatoren eignen sich als Iterator in einer Klasse

```
...
def __iter__(self):
    ...
    yield x
```

- Generatoren sind meistens schneller und speichereffizienter
- Iteratoren sind flexibler, als z.B. Listen

Funktionen höherer Ordnung I

- Alles in Python ist ein Objekt, auch Funktionen und Klassen
- Objekte kann man als Funktionsargument übergeben

```
def say_hi():
    return "hi"

def make_louder( f ):
    return f().upper()

shout_hi = make_louder(say_hi)
say_hi()    # "hi"
shout_hi()  # "HI"
```

Funktionen höherer Ordnung II

- Ebenso können Funktionen von Funktionen zurückgegeben werden

```
def first_n_squares( n : int ):
    def print_squares():
        print([i*i for i in range(1,n+1])
    return print_squares

first5 = first_n_squares(5)
first5() # [1, 4, 9, 16, 25]
```

Decorator I

- Dekorateure erwarten eine Funktion als Argument und geben eine Funktion zurück
- Um eine Funktion zu dekorieren schreibt man ***@dec*** in die Zeile über der Definition
- Diese Syntax ist gleichbedeutend mit $f = dec(f)$

Decorator II

```
def make_louder( f ):
    def wrapper(*args):
        print("In the Wrapper")
        result = f(*args)
        return result.upper()
    return wrapper

@make_louder
def say_hi():
    return "hi"

say_hi()      # Prints "In the Wrapper"
              # Returns "HI"
```

Nützliche Decorators I

Beispiel (wraps, total_ordering)

```
from functools import wraps, total_ordering
def mydec(f):
    @wraps(f)
    def wrapper(*args, **kwargs): ...
    return wrapper

@total_ordering
class Human:
    def __eq__(self, other): ...
    def __gt__(self, other): ...
```

Nützliche Decorators II

Beispiel (contexmanager)

```
from contextlib import contextmanager
@contextmanager
def get_score(path):
    scorefile = ...
    yield scorefile
    scorefile.close()

with get_score() as scores:
    scores.add( entry )
```

Nützliche Decorators III

Beispiel (lru_cache)

```
from functools import lru_cache
@lru_cache(maxsize=2)
def fib(n):
    if n <= 1:
        return n
    return fib(n-2) + fib(n-1)
```

Quellen und weiterführende Informationen I

Handout und Beispiele als Jupyter-Notebooks

`https://mybinder.org/v2/gh/janDigeser/PythonSeminar/master`

Abstract Base Class Documentation

`https://docs.python.org/3/library/abc.html`

Lamdba Funktionen und Beispiele

`https://www.python-kurs.eu/lambda.php`

Iteratoren und Generatoren Unterschiede

`https://nvie.com/posts/iterators-vs-generators/`

Dokumentation zum yield-Statement

`https://docs.python.org/3/reference/expressions.html#yieldexpr`

PEP über Generatoren

`https://www.python.org/dev/peps/pep-0342/`

Quellen und weiterführende Informationen II

Schnelle Einführung für Dekorateure

`http://jfine-python-classes.readthedocs.io/en/latest/decorators.html`

Weiterführendes zu Dekorateure

`https://www.artima.com/weblogs/viewpost.jsp?thread=240808`

`https://www.artima.com/weblogs/viewpost.jsp?thread=240845`