Leibniz Universität IT Services

Leibniz Universität Hannover

Python

„Iteratoren und Generatoren“

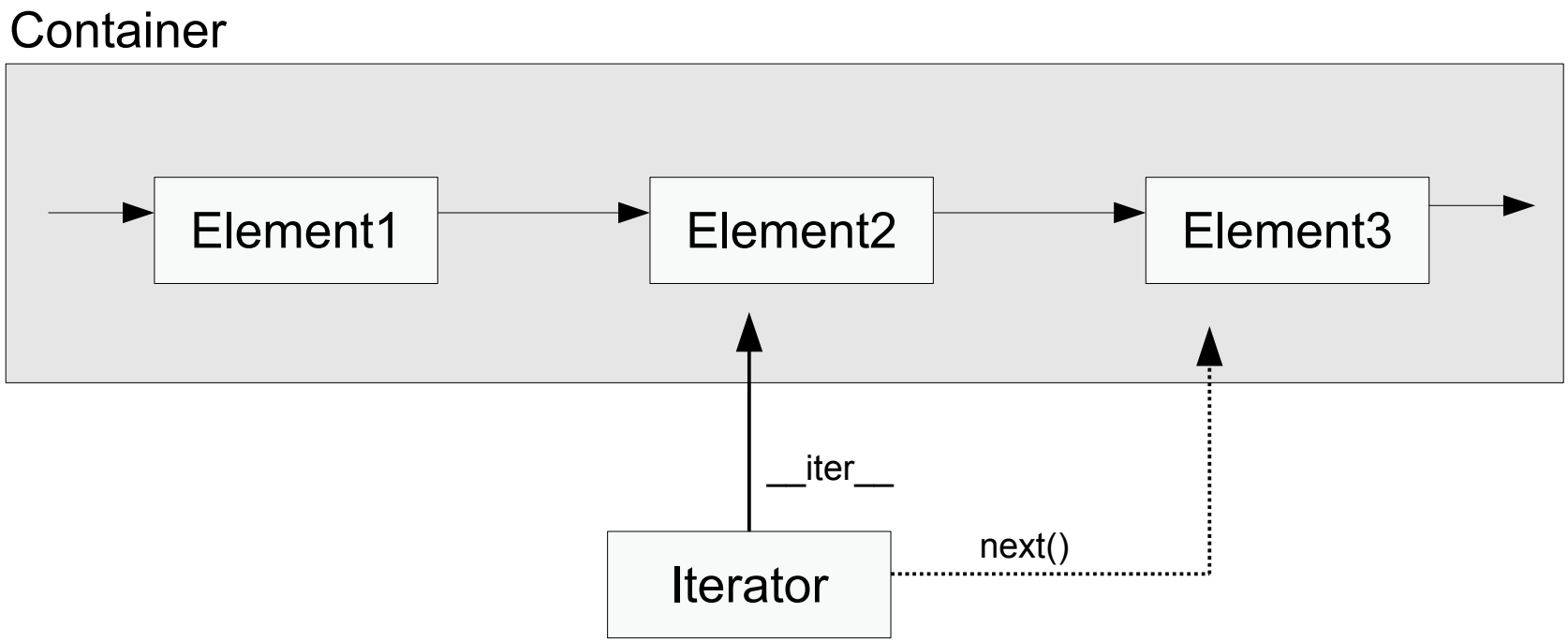
Iterierbare Objekte

- Container-Objekte, die ihre Elemente nacheinander zurückgeben können. Zum Beispiel `file` und `dict`.
- Sequenzen wie `list`, `str` und `tuple`.
- Objekte, die die Methoden `__iter__()` oder `getitem()` implementiert haben. Die Methode `__iter__()` liefert einen Iterator zurück. Die Methode `getitem()` nutzt Indizes, beginnend bei 0.
- Objekte, deren Elemente mit Hilfe einer for-Schleife durchlaufen werden können.

Iteratoren

- Standardschnittstelle für den Zugriff auf die Elemente eines iterierbaren Objekts.
- Objekt, welches eine `__next__()` - Methode implementiert. Diese Methode gibt das nächste Element zurück. In einer for-Schleife wird diese Methode automatisiert aufgerufen.
- Wenn kein Element mehr vorhanden ist, gibt das Objekt die Ausnahme `StopIteration` aus.

Arbeitsweise



Nutzung in einer foreach-Schleife

```
liste = [1, 2, 3, 4]

for element in liste:
    print(element)
```

Erläuterung

- Das iterierbare Objekt steht rechts vom Schlüsselwort `in`.
- Dieses Objekt wird Element für Element durchlaufen.
- Bei jeden Durchlauf wird das aktuelle Element automatisiert durch den Iterator aufgerufen.

Bau eines eigenen Iterators

```
liste = [1, 2, 3, 4]
lst_iter = iter(liste)

elementVorhanden = True

while elementVorhanden:
    try:
        element = lst_iter.__next__()
        print(element)

    except StopIteration:
        elementVorhanden = False
```

Definition eines Iterators

```
lst_iter = iter(liste)
```

- Der Funktion iter() wird ein iterierbares Objekt als Argument übergeben. Das Argument soll Element für Element durchlaufen werden.
- Die Funktion selber gibt einen Iterator zurück.

Nächstes Element

```
element = lst_iter.__next__()
```

- Der Iterator wird mit der Methode __next__() durch den Punktoperator verbunden.
- Mit Hilfe der Methode __next__() des Iterators wird das nächste Element geholt.

Exception

```
except StopIteration:
        elementVorhanden = False
```

- Falls keine Elemente mehr in dem iterierbarem Objekt vorhanden sind, wird die Ausnahme StopIteration ausgelöst.
- Mit Hilfe von try… except muss diese Ausnahme abgefangen werden.

Fehlerbehandlung (Exception-Handling)

try: element = lst_iter.__next__() print(element)	Versuche die Anweisungen auszuführen
except StopIteration: print("Ende foreach")	Abfangen von benannten Ausnahmen / Fehlern
except: print("x beliebiger Laufzeitfehler")	Abfangen von allen anderen Fehlern
else: print("Durchführung der Anweisung")	Wenn kein Fehler aufgetreten ist ...
finally: print("Immer ausführen")	Diesen Block immer ausführen

… in einer eigenen Klassen nutzen

```
class clsGnom(object):

    def __init__(self, dateiName, pfad = None):
        pass

    def __iter__(self):
        pass

    def __next__(self):
         pass
```

Iterierbares Objekt

```
class clsGnom(object):

    def __init__(self, dateiName, pfad = None):
        self._basenfolge = []
        self.posSequenz = -1
        self.posElement = -1

        speicherort = pfad + "/" + dateiName
        objDatei = open(speicherort, 'r')

        for strZeile in objDatei.readlines():
            self._basenfolge.append(strZeile)

        objDatei.close()
```

Erläuterung

- In der `__init__()` - Methode wird das iterierbare Objekt `File` in der for-Schleife genutzt.
- Die Zeilen der Datei werden in einer Liste gespeichert. Diese Liste wird mit Hilfe eines Iterators durchlaufen.

Implementierung der Methode __iter__

```
def __iter__(self):
    return self
```

- Die Methode wird automatisiert aufgerufen, wenn die Instanz der Klasse mit Hilfe einer for-Schleife durchlaufen wird.
- Wenn diese Methode überschrieben wird, muss auch die Methode __next__() implementiert werden. Andernfalls wird der Fehler *TypeError: iter() returned non-iterator of type* angezeigt.

Überschreiben der Methode __next__

```
def __next__(self):
        laenge = len(self._basenfolge[self.posSequenz])
        if (self.posElement + 1) >=   laenge):
            self.posElement = 0
            self.posSequenz +=1
        else:
            self.posElement +=1

        if self.posSequenz >= len(self._basenfolge):
            raise StopIteration()

        return self._basenfolge[self.posSequenz][self.posElement]
```

Erläuterung

- Die Methode .__next__ erzeugt einen Verweis auf das nächste Element.
- Die vorhandene Methode wird überschrieben, um jedes Element einer zweidimensionalen Liste zu durchlaufen. Die erste Dimension enthält die eingelesenen Zeilen. Die zweite Dimension die einzelnen Zeichen pro Zeichen. Sobald alle Zeichen einer Zeile gelesen wurden, wird in die nächste Zeile gewechselt.
- Die Methode gibt die Ausnahme StopIteration() zurück, wenn alle Zeilen und Zeichen gelesen sind.

Nutzung in einer for-Schleife

```
meinGnom = clsGnom("zitterrochen.txt")

for element in meinGnom:
    print(element)
```

- Im ersten Schritt wird eine Instanz erzeugt. Diese Instanz ist ein Container für x Zeilen aus einer Datei.
- Dieser Container wird mit Hilfe der for-Schleife zeichenweise durchlaufen. Sobald die Methode `__next__()` die Ausnahme `StopIteration` ausgibt, wird die for-Schleife abgebrochen.

Nutzung in einer while-Schleife

```
meinGnom = clsGnom("zitterrochen.txt")

sequenzVorhanden = True

while sequenzVorhanden:
    try:
        sequenz = meinGnom.__next__()
        print(sequenz)
    except StopIteration:
        sequenzVorhanden = False
```

Erläuterung

- Im ersten Schritt wird eine Instanz erzeugt. Diese Instanz ist ein Container für x Zeilen aus einer Datei.
- Anschließend wird eine Variable definiert. Solange die boolsche Variable den Wert `True` (Wahr), wird die Schleife durchlaufen.
- Der try-Block fasst die auszuführenden Anweisungen zusammen. Bei jedem Schleifendurchlauf soll eine Referenz auf das nächste Element mit Hilfe der Methode `__next__()` gespeichert werden.
- Falls diese Methode die Ausnahme `StopIteration` ausgibt, wird die Schleife abgebrochen.

Filterung von Elemente

- Mit Hilfe einer benutzerdefinierten Funktion.
- Nutzung der filter()-Funktion.
- Nutzung von Generatoren.

Eigene Filter-Funktion

```
def count_Sequenz(self, sequenz):
    lstFilter = []

    for element in self._basenfolge[self.posSequenz]:
        if (element == sequenz):
            lstFilter.append(element)

    return len(lstFilter)
```

Nutzung der Filter-Funktion

```
def count_Liste(self, sequenz):
    iterFilter = filter(lambda x: x == sequenz,
                        self._basenfolge[self.posSequenz])

    return len(list(iterFilter))
```

filter-Funktion

```
filter(lambda x: x == sequenz, self._basenfolge[self.posSequenz])
```

- Der Funktion filter() filtert Elemente in Abhängigkeit einer Bedingung aus einer Liste.
- Als erster Parameter wird eine Funktion angegeben. Diese Funktion bildet das Filterkriterium ab.
- Der zweite Parameter definiert das iterierbare Objekt. Dieses Objekt wird entsprechend der Filterkriterium durchsucht.
- Die Funktion gibt einen Iterator als Wert zurück.

2. Parameter

```
filter(lambda x: x == sequenz, self._basenfolge[self.posSequenz])
```

- Welches iterierbares Objekt soll gefiltert werden?
- In diesem Beispiel wird nach einem Zeichen in einer Zeile gesucht.

1. Parameter

```
filter(lambda x: x == sequenz, self._basenfolge[self.posSequenz])
```

- Der erste Parameter legt das Filterkriterium fest.
- In diesem Beispiel wird das Filterkriterium mit Hilfe einer lambda-Funktion festgelegt. Die Funktion gibt das Ergebnis des Vergleiches zurück.

lambda-Funktion

lambda	x	:	x == sequenz
lambda	Argumentliste	:	Ausdruck

- Die Funktion beginnt mit dem Schlüsselwort lambda.
- Die Funktion besitzt keinen Funktionsnamen. Die Funktion ist anonym.

Argumentliste

lambda	x	:	x == sequenz
lambda	Argumentliste	:	Ausdruck

- Einer lambda-Funktion können beliebig viele Argumente übergeben werden.
- Die einzelnen Argumente werden durch ein Komma getrennt.

Anweisung

lambda	x	:	x == sequenz
lambda	Argumentliste	:	Ausdruck

- Jede anonyme Funktion hat nur eine Ausdruck.
- In diesem Beispiel wird das übergebene Argument mit einem Wert verglichen.
- Die Funktion gibt einen Wert entsprechend des Ausdrucks zurück.

Mappen von Listen

- Mit Hilfe einer eigenen Funktion wird aus einem iterierbaren Objekt ein neues Objekt erzeugt.
- Mit Hilfe der Funktion `map()` erzeugt aus einer bestehenden Liste eine neue Liste.

Eigene Funktion

```
def shiftCharacterToLower(self):
    lstRow = []

    for zeichen in self._basenfolge[self.posSequenz]:
            if zeichen.islower() == False:
                lstRow.append(chr(ord(zeichen) + 32))
            else:
                lstRow.append(zeichen)



    return lstRow
```

Nutzung der map-Funktion

```
def shiftBuchstabeToLower(self):
    iterMap = map(lambda  x :
                        chr(ord(x) + 32) if x.islower() == False else x ,
                        self._basenfolge[self.posSequenz])

    return list(iterMap)
```

map-Funktion

```
iterMap = map(lambda  x :
                    chr(ord(x) + 32) if x.islower() == False else x ,
                    self._basenfolge[self.posSequenz])
```

- Der Funktion map() bearbeitet aus einer Liste Elemente und speichert diese wieder in einer Liste.
- Als erster Parameter wird eine Funktion angegeben. Diese Funktion wird auf alle Elemente des Objekts angewendet. Jedes Element wird durch die Funktion neu berechnet.
- Der zweite Parameter definiert das iterierbare Objekt. Dieses Objekt wird als Grundlage für die Berechnung genutzt.
- Die Funktion gibt einen Iterator als Wert zurück.

lambda-Funktion

lambda	x	:	chr(ord(x) + 32) if x.islower() == False else x
lambda	Argumentliste	:	Ausdruck

- Die Funktion beginnt mit dem Schlüsselwort lambda.
- Die Funktion besitzt keinen Funktionsnamen. Die Funktion ist anonym.
- Die anonyme Funktion gibt immer einen Wert zurück.

Argumentliste

lambda	x	:	chr(ord(x) + 32) if x.islower() == False else x
lambda	Argumentliste	:	Ausdruck

- Einer lambda-Funktion können beliebig viele Argumente übergeben werden.
- Die einzelnen Argumente werden durch ein Komma getrennt.

Anweisung

lambda	x	:	chr(ord(x) + 32) if x.islower() == False else x
lambda	Argumentliste	:	Ausdruck

- Jede anonyme Funktion hat nur eine Ausdruck.
- In diesem Beispiel wird eine if – else – Anweisung in der Form Ausdruck if Bedingung Else Ausdruck nachgebildet.
- Wenn die Bedingung x.islower() == False wahr ist, wird der Ausdruck chr(ord(x) + 32) ausgeführt. Andernfalls wird der Ausdruck x angewendet.

Generatoren

- Erzeugung von Iteratoren.
- „just-in-time“-Konstruktion eines Iterators. Die Funktionen __iter__() und __next__() sind automatisch implementiert.
- Nutzung bei sehr großen Listen.

Beispiel

```
def shiftGeneratorToLower(self):
        for zeichen in self._basenfolge[self.posSequenz]:
                if zeichen.islower() == False:
                    yield (chr(ord(zeichen) + 32))

                else:
                    yield zeichen
```

Schlüsselwort yield

```
def shiftGeneratorToLower(self):
        for zeichen in self._basenfolge[self.posSequenz]:
                if zeichen.islower() == False:
                    yield (chr(ord(zeichen) + 32))

                else:
                    yield zeichen
```

- Durch das Schlüsselwort yield wird die Methode angehalten.
- Die Kontrolle wird an den Aufrufer zurückgegeben.

Aufruf der Funktion

```
for element in meinGnom.shiftGeneratorToLower():
    print(element)
```

- Die Generator-Methode wird wie jede andere Methode über die Instanz aufgerufen.
- Mit Hilfe des Namens wird ein Generator aufgerufen.
- Die Methode gibt einen Iterator zurück.

Arbeitsweise

- Bei jedem Schleifendurchlauf wird die Generator-Methode bis zum yield abgearbeitet.
- Mit Hilfe von yield wird ein Wert an den Aufrufer zurückgegeben. Lokale Variablen und deren Status bleibben erhalten. Der Aufrufer kann den nächsten Schleifendurchlauf startet.
- Sobald keine Elemente mehr vorhanden sind, wird automatisiert StopIteration angezeigt.