Scripting und Algorithmen

Python II

Harald Lampesberger

FH OÖ, Department für Sichere Informationssysteme
SAL2VO, SS23, Version: 19. April 2023

FH OBERÖSTERREICH
UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

Hash-Datenstrukturen

Hash-Funktion

Lineare Datenstrukturen reichen manchmal nicht aus

- Wir können Objekte über ihre Position (int) indizieren
- Generelles Mapping von einem Key $\rightarrow$ Value, wo der Key kein Integer ist?

Idee: Wir nutzen eine Hash-Funktion

- Values werden in einer linearen Datenstruktur abgelegt (mit Integer-Indices)
- Position: $hash(key) \mod arraysize$
- Hash-Funktion kann mit strings, int, float, usw umgehen
- Die Berechnung eines Hashes ist typischerweise in $\mathcal{O}(1)$

Beispiel mit Array-Größe 10

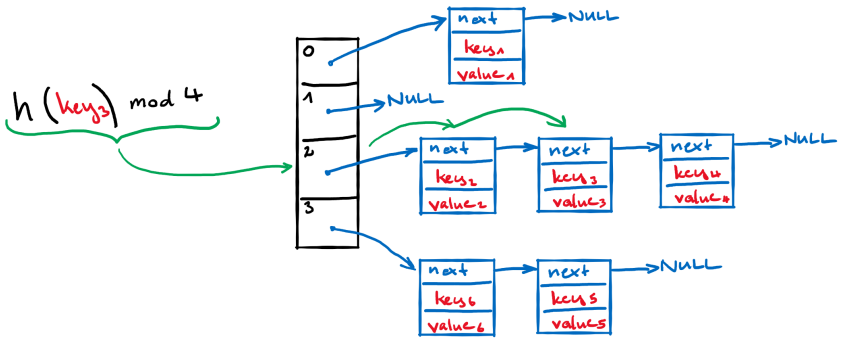
```
index1 = hash("Hello World") % 10       # 0
index2 = hash("Hello World!") % 10      # 5
```

Hash-Kollision und Auflösung

Unterschiedliche Keys mit gleichem Hash = Kollision

- Direkte Nutzung eines Arrays ist daher eine schlechte Idee
- Verschiedene Ansätze zur Kollisionsauflösung

Einfaches Beispiel: Hashtable mit verketteten Listen



Hash-Datenstrukturen mit Python

set I/II

Eine Menge (**set**) ist eine ungeordnete veränderliche Hash-Datenstruktur

- Objekte müssen (!!) unveränderlich sein, da sich sonst der Hashwert ändern könnte
- Typen in einem Set können beliebig gemischt sein; aber keine Duplikate

```
s = set([3, 5, 10])  # set of numbers
s = {3, 5, 10}       # alternative syntax
t = set("Hello")     # t = {'H', 'e', 'l', 'o'}
u = {"Hello"}        # u = {'Hello'}
```

Sets haben eine Kardinalität: **len**(t)

Leere Menge: s = **set**()

Iteration mittels for-Schleife

```
for elem in t:
    print(t)
```

set II/II

Mengenoperationen

```
a = t | s  # Union a = {'H', 'e', 'l', 'o', 3, 5, 10}
b = t & s  # Intersection b = {}
c = t - s  # Set difference
d = t ^ s  # Symmetric difference (XOR)
```

Elemente hinzufügen/entfernen

```
t.add('x')              # Add single item
s.update([10, 37, 42]) # Add multiple items
t.remove('H')           # Remove single item
res = 'x' in t          # res = True
```

frozenset

frozenset ist eine ungeordnete unveränderliche Hash-Datenstruktur

- Unveränderliche Variante von **set**
- Wird beim Erzeugen im Konstruktor durch ein iterable befüllt

```
s = frozenset([3, 5, 10])  # set of numbers
t = frozenset("Hello")     # t = {'H', 'e', 'l', 'o'}
```

Mengenoperationen sind möglich

- Jede Mengenoperation erzeugt ein neues frozenset-Objekt

```
s | t        # returns frozenset({'e', 3, 5, 'H', 'l', 10, 'o'})
```

Warum? unveränderlicher Key in einem dict, Sets of Sets

dict I/II

Ein Dictionary (**dict**) ist ein veränderliches Mapping (Key-Value-Speicher)

- Implementiert mittels Hash-Datenstruktur
- Zu einem Key-Objekt wird ein Value-Objekt abgespeichert
- Key-Objekte müssen (!!) unveränderlich sein, da sich sonst der Hashwert ändern könnte
- Keine Einschränkungen bezüglich Typen

```
stock = {"name"   : "GOOG",
         "shares" : 100,
         "price"  : 490.10}
```

Leeres Dictionary: `d = dict(); d = {}`

Zugriff mittels Subscript-Operator und Key-Objekt

```
v = stock["price"]
print(v)
print(stock["name"])
```

dict II/II

Objekte einfügen, verändern, löschen

```
stock["shares"] = 75
stock["date"] = "June 7, 2007"
del(stock["name"])
```

Key testen: `"name" in stock` `# True`

Iteration mittels for-Schleife

```
for k in stock.keys():
    print(k)       # shares, price, date
for v in stock.values():
    print(v)       # 75, "June 7, 2007", 490.10
for k, v in stock.items():
    print(k, v)
```

Module

Module

Der ganze Code in einer Datei wird unübersichtlich, deswegen Modularisierung

```
# File: tools.py
def divide(a, b):
  return a / b
```

Objekte aus einem Modul laden auf drei Varianten

```
import tools              # Original Namespace (empfohlen)
import tools as t         # Umbenannter Namespace (empfohlen)
from tools import *       # Import in den eigenen Namespace (Konflikt?)
from tools import divide # Alternative in den eigenen Namespace

x = tools.divide(1.0, 2.0)
x = t.divide(1.0, 2.0)
x = divide(1.0, 2.0)
```

Module Installieren

- `ImportError` falls Modul nicht im Pfad gefunden wird
- Nachinstallieren direkt in einer Shell:
  - `conda install` *modulename*
  - oder `pip install` *modulename*
  - zum Beispiel: `pip install pyparsing`

Beispiel: deque

Eine Deque (`collections.deque`) ist eine veränderliche Queue-Struktur

- „double-ended queue“
- Implementiert als doppelt-verkettete Liste
- Objekte können am Anfang und Ende sehr schnell eingefügt/entfernt werden ($\mathcal{O}(1)$)

```
from collections import deque

d = deque()
d.append(1)
d.append(2)
d.append(3)
d.popleft()     # returns 1
d.popleft()     # returns 2
d.popleft()     # returns 3
```

Klassen, Exceptions

Klassen I/II

Klassendefinition mittels **class**

- **self** ist die Selbstreferenz, vgl. this in Java/C++
- Erstes Argument einer Methode ist immer **self**
- *__methode__* kennzeichnet Spezialfunktionen für Operator-Overloading

```
class Stack(object):                # In Klammern, erbt alles von object
    def __init__(self):             # Constructor
        self.store = []             # Member, immer public

    def push(self, obj):            # Methode mit einem Argument
        self.store.append(obj)

    def pop(self):                  # Methode ohne Argument
        return self.store.pop()

    def __len__(self):              # wird bei len() aufgerufen
        return len(self.store)
```

Klassen II/II

Lebenszyklus

```
s = Stack()          # Call-Operator mit dem Klasse instantiiert ein Objekt
s.push(1)
s.push(2)
print(len(s))        # 2
s.pop()              # returns 2

del(s)               # Loescht Variable s;
                     # ohne Referenz wird Stack-Objekt vom GC freigegeben
```

Vererbung und Reservierte Methoden

Stack ist ähnlich wie eine Liste → Vererbung und Overrides

```
class Stack2(list):
    def push(self, obj):
        self.append(obj)
```

Reservierte Methodennamen

- `__repr__(self)` für `repr(obj)`
- `__str__(self)` für `str(obj)`
- `__len__(self)` für `len(obj)`
- `__add__(self, other)` für `s + t`
- `__mul__(self, other)` für `s * t`
- `__eq__(self, other)` für `s == t`
- uvm... siehe Python-Dokumentation

Exceptions

Exceptions fangen

```
a = []
try:
    r = a.pop()
except IndexError as e:
    print(e )
finally:        # finally-Block ist optional; wird IMMER ausgefuehrt
    r = -1      # auch wenn keine Exception auftritt
```

Sind auch nur Objekte

```
class MyException(Exception):
    pass

if error_appeared:
    raise MyException("Something is wrong")
```

`str()` vs. `repr()`

Zwei Varianten, um Objekt in String zu verwandeln

- Repräsentation in py-Syntax: **`repr`**`(obj)` called `obj.__repr__()`
- Lesbarer String, zB für print(): **`str`**`(obj)` called `obj.__str__()`
- `obj.__str__` **`is`** `obj.__repr__` falls nicht überladen

Best practice

- `obj.__repr__()` soll valide Python-Syntax ausgeben
- `obj.__str__()` für „schöne“ Ausgabe

```
class X(object):
    def __str__(self):
        return "very nice X"
    def __repr__(self):
        return "X()"
x = X()
repr(x)              # 'X()'
str(x)               # 'very nice X'
print(x)             # very nice X
print(1, x)          # 1 very nice X
print([x])           # [X()]
```

Comprehensions und Generatoren

Comprehensions

Deklarative Beschreibung wie in der Mathematik

- zB $S = \{2, 3, 5, 7, 9, 11\}$ und $T = \{x \mid 1 \leq x \leq 5 \text{ und } x \in S\}$

```
s = {2, 3, 5, 7 ,9, 11}
t = {x for x in range(1, 6) if x in s}
# t = set([2, 3, 5])

u = [b*a for a in range(1, 3) for b in t if b < 5]
# u = [2, 3, 4, 6]

v = {k: list(range(k)) for k in u}
# v = {2: [0, 1],
#      3: [0, 1, 2],
#      4: [0, 1, 2, 3],
#      6: [0, 1, 2, 3, 4, 5]}
```

Generatoren

Generatoren sind ein zentrales Konzept in Python 3.x

- Schlüsselwort **yield** in einer Funktion
- Generator-Comprehension

Ein Generator verhält sich wie ein Iterator

- yield im Funktionsblock verwandelt die Funktion in einen Generator (Vgl. Co-Routine)
- **range**() ist zB ein Generator

```
def lazygen(strng):
    for char in strng:
        yield f"lazy ... {char}"

for s in lazygen("abc"):          # Anwendung
    print(s)
```

Generator-Comprehension

Generator-Comprehension mit runden Klammern

- Konzeptionell wie eine list-Comprehension, aber ohne eckige Klammern
- Als einzelnes Argument können die runden Klammern sogar weggelassen werden

```
>>> (x*2 for x in range(3))
<generator object <genexpr> at 0x0000000003220C60>
>>> sum(x*2 for x in range(3))
6
>>> list(x*2 for x in range(3))
[0, 2, 4]
```

File I/O

File I/O

open öffnet ein File und liefert ein Handle-Objekt zurück

- Vieles kann mit Files schief gehen
- Unterscheidung: text (r/w, default) und binary (rb/wb)
- Bei text wird Encoding von der Plattform übernommen, kann aber auch angegeben werden

`with as` Statement kümmert sich um kontrollierte Ausführung

- zB File schließen nach Ende des Blocks

```
with open("/tmp/data.txt", "r") as fd:
  for line in fd:               # Default-Iterator Textzeilen
    print(line)

with open("/tmp/data.txt", "rb") as fd:
    bin_data = fd.read(64)      # lese bis zu 64 Bytes

with open("data.txt", "w") as fd:
    chars_written = fd.write("Hello World\n")
```