Flik 9: Avancerat

Iterabler och iteratorer

- Många funktioner returnerar *iterabler*, t.ex. `range`, `map` m.fl.
  - dvs. de går att iterera över, eller *lazy evaluation*
- *iter(iterabel)* returnerar en iterator
- *next(iterator)* returnerar nästa värde eller undantaget StopIteration

```
>>> myTuple = (123, 'xyz', 45)
>>> i = iter(myTuple)
>>> next(i)
123
>>> next(i)
'xyz'
>>> next(i)
45
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iteratorer

- `range` returnerar en iterabel

```
>>> b = iter(range(1000000000000000))
>>> next(b)
0
>>> next(b)
1
```

Iteratorer

- for-loopar använder iteratorer bakom kulisserna

```
for i in seq:
    do_something_to(i)

#Bakom kulisserna:

iterator = iter(seq)
while 1:
    try:
        i = next(iterator)
    except StopIteration:
        del iterator
        break
    do_something_to(i)
```

Iteratorer

- Iterera över dictionaries

```
for k in my_dict:
    print('Key:', k)
    print('Val:', my_dict[k])

# Alt
k = iter(my_dict.keys())
v = iter(my_dict.values())
i = iter(my_dict.items())

next(k)
next(v)
next(i)
```

Iteratorer

- Iterera över filer

```
my_file = open('fil.txt')

for each_line in my_file:
    print(each_line, end='')
```

Egna iterabler och iteratorer

- Python exekverar iter(s) som s.__iter__()
- Python exekverar next(i) som i.__next__()
- En iterabel är ett objekt av en klass som implementerar __iter__
- En iterator är ett objekt av en klass som implementerar __next__
- En iterator bör också implementera __iter__ som ska returnera self.
- Ibland används samma klass som både iterator och iterabel, men det kan hindra att man har mer än en oberoende iterator åt gången.

Iteratorer

- Utöka en klass med en iterator

```
from random import choice
class RandSeq(object):
    def __init__(self, seq):
        self.data = seq
    def __iter__(self):
        return self
    def __next__(self):
         return choice(self.data)

>>> rs = RandSeq(('sten', 'sax', 'påse'))
>>> for each_item in rs:
        print(each_item)
sax
sten
påse
...
```

Generatorfunktioner

- *yield()*
  - En generatorfunktion är en funktion som innehåller *yield*
  - Generatorfunktionen returnerar en *generator* (även kallat ett *generatorobjekt*)
  - Generatorobjekt fungerar som iteratorer.
  - Generatorn pausar exekveringen vid varje yield som *överlämnar* nästa värde

```
def simple_gen():
    yield 1
    yield 'kalle'

>>> my_g = simple_gen()
>>> next(my_g)
1
>>> next(my_g)
'kalle'
>>> next(my_g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Exempel: Generatorfunktion

- En egen version av `range`

```
def my_range(maximum):
    i = 0
    while i < maximum:
        yield i
        i += 1

>>> list( my_range(7) )
[0, 1, 2, 3, 4, 5, 6]
>>> for x in my_range(3):
...     print(x)
...
0
1
2
>>>
```

Fibonaccisekvensen

- Börja med 0 och 1, sedan ska varje nytt tal vara summan av de två senaste

```
def fibonacci(max):
    detta = 0
    nästa = 1
    while detta <= max:
        yield detta
        detta, nästa = (nästa, detta+nästa)


>>> list(fibonacci(100))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>
```

Delegera till subgenerator

- Python 3.3 introducerade **yield from** som delegerar till en subgenerator.
- Om generatorn anropar return, så returneras värdet via yield from.

```
def kedja():
    yield from my_range(3)
    yield from fibonacci(6)

>>> for item in kedja():
...     print(item)
0
1
2
0
1
1
2
3
5
```

Utökade generatorer

- *send()* – skickar ett värde till generatorn.
  - Värdet ”returneras” av yield till generatorn.
  - Generatorn måste alltså ”preppas” med en next först.
- *close()* – avbryter generatorn
- *throw()* – genererar ett undantag inuti generatorn
- Möjliggör dubbelriktad kommunikation – *coroutine*

Utökade generatorer

```
def genomsnitt():
    antal, summa, snitt = 0, 0.0, 0.0
    while True:
        x = yield snitt
        if x is not None:
            summa += x
            antal += 1
            snitt = summa / antal
```

```
>>> s = genomsnitt()
>>> next(s)   # preppa!
0.0
>>> s.send(7)
7.0
>>> s.send(3)
5.0
>>> s.send(2)
4.0
```

Generator expressions

- Samma syntax som list comprehensions fast med parenteser (...)
  - `(expr for iter_var in iterable)`
  - `(expr for iter_var in iterable if cond_expr)`
- Sparar minne - behöver inte ha alla data innan evaluering
- Exempel – generera rader som matchar ett reguljärt uttryck:

```
import re

fh = open('messages', 'r')
pat = re.compile('kernel: (.*)')
kmsg = ( pat.search(x).group(1)
         for x in fh if pat.search(x) )

for msg in kmsg:
    print(msg)

fh.close()
```

Övning

- **ovn0901/iteratorklass.py** Skapa en iteratorklass med namnet `it` som varje gång `next` anropas ökar med ett givet steg. Första iterationen ska returnera 0. T.ex.

```
>>> a = iter(it(0.1))
>>> next(a)
0.00000..
>>> next(a)
0.10000..
>>> next(a)
0.20000..
```

- **ovn0902/generatorfunk.py** Skapa en generatorfunktion med namnet `Stega` som gör samma sak som iteratorn i föregående övning.

```
>>> a = Stega(0.1)
>>> print(next(a), next(a), next(a))
0 0.1 0.2
>>>
```

Övning

- **ovn0903/kvadratrot.py*** Skapa funktionerna `f1`, `f2` och `f3`. De ska alla ta ett positivt heltal som parameter och de ska returnera en sekvens av kvadratroten av alla heltal (`math.sqrt`) från 1 upp till (men inte med) det givna heltalet.
  - `f1` ska använda `map`
  - `f2` ska använda en list comprehension
  - `f3` ska använda en generator expression

asyncio

- I Python 3.5 introducerades ny syntax för korutiner
  - async def
  - await
  - async with
  - async for
- Bygger internt på utökade generatorer
- Kräver en event loop som driver korutinerna
- Standardmodulen asyncio implementerar event loop m.m.
- Liknande syntax finns i en del andra programmeringsspråk
- Hanterar olika uppgifter samtidigt i en enda exekveringstråd genom att varje uppgift frivilligt lämnar över kontrollen när den inte har något att göra.
  - Behöver inte tänka på ”trådsäkerhet” eftersom det är enkeltrådat.
  - Liknar node.js, men man slipper callback-funktioner!
  - Passar bättre vid I/O-intensiva uppgifter än CPU-intensiva.
  - Nätverkskommunikation, databashantering osv.

Simultanschack?

Exempel, asyncio

```
import asyncio, sys, time

async def show_time():
    start = time.time()
    d = ""
    while True:
        t = "{:6.1f}".format(time.time()-start)
        sys.stdout.write(d+t)
        sys.stdout.flush()
        d = chr(8) * len(t)
        try:
            await asyncio.sleep(0.1)
        except asyncio.CancelledError:
            break
```

Exempel, forts.

```
async def beräkning():
    await asyncio.sleep(5)
    return 1+1

async def program():
    task = asyncio.ensure_future(show_time())
    res = await beräkning()
    task.cancel()
    await task
    print(" resultat:", res)

loop = asyncio.get_event_loop()
loop.run_until_complete(program())
loop.close()
```

Trådning - threading

- Skapa en klass som ärver av klassen *threading.Thread*
- Skapa en metod *run()* som själva exekveringen kommer äga rum i
- Skapa en instans av denna klass och starta tråden

```
class MyThread(threading.Thread):
    def run(self)
        #do things

my_th = MyThread()
my_th.start()
```

Echoserver: huvudprogram

```
import socket, queue
from clienthandler import Client  # se nästa sida!
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 1337))                # Lyssna på port 1337
s.listen(50)

done = queue.Queue()              # Klienter som ska tas bort
clients = {}                      # Alla klienter lagras här
while True:
    conn, caddr = s.accept()      # caddr är klientens ip+port
    clients[caddr] = Client(conn, caddr, done)
    clients[caddr].start()        # klienten kör i egen tråd
    while True:
        try:                      # städa bort klara klienter
            caddr = done.get(block=False)
            clients[caddr].join()
            del clients[caddr]
        except queue.Empty:
            break
```

Echoserver: En tråd per klient

```
import threading, queue

class Client(threading.Thread):
    def __init__(self, conn, caddr, q):
        super().__init__()
        self._conn = conn
        self._caddr = caddr
        self.done_queue = q
    def run(self):
        try:
            while True:
                msg = self._conn.recv(2000)
                if not msg: break
                self._conn.sendall(msg)
        except Exception:
            pass
        finally:
            self._conn.close()
            self.done_queue.put(self._caddr)
```

Trådning - threading

- Huvudproblem med trådning - delade resurser "kritiska sektioner"
- Enbart en tråd i taget ska accessa dessa kritiska sektioner

```
class Counter:
    def __init__(self):
        self.lock = threading.Lock()
        self.value = 0

    def increment(self):
        self.lock.acquire() # critical section
        self.value = value = self.value + 1
        self.lock.release()
        return value
```

Trådning - threading

- Ännu bättre - använd *try: ... finally:* - släpper låset även om något går fel

```
lock.acquire()
try:
    # minimum av operationer som kräver låsning
finally:
    lock.release()
```

- Eller bäst:

```
with some_lock:
    print("some_lock is locked here")
```

Processhantering - subprocess

- Modulen *subprocess* har hjälpfunktioner för att exekvera program
  - `subprocess.run(['ls', '-la'])`
- Fånga utskriften

```
import subprocess

try:
    ret = subprocess.run(['/bin/ls', '-la'],
                         capture_output=True,
                         encoding='utf-8')
    if ret.returncode == 0:
        print(ret.stdout, end='')
    else:
        print('Misslyckades:', ret.stderr, end='')
except (OSError, UnicodeDecodeError) as e:
    print(e)
```

Processhantering - modulen os

- Övrig processhantering finns i modulen `os` och kan vara plattformsberoende
- Lågnivå - ofta med direkta motsvarigheter i C
  - `os.fork()`
  - `os.nice()`
  - `os.kill()`
  - m.fl
- Använd företrädesvis modulen `subprocess`.