14 Nyttige innebygde funksjoner og mekanismer i Python

14.1 Læringsmål dette kapittel

<< kommer >>

14.2 Innledning

Python har mange innebygde funksjoner og mekanismer som er nyttige for å skrive effektiv, «pythonic» kode. I dette kapittelet ser vi på noen av de mest allsidige og nyttige funksjonene og mekanismene du bør kjenne til, og som vi ikke har omtalt spesielt tidligere.

Merk:

I flere av funksjonsbeskrivelsene nevnes mekanismen ***generator***. Det kan være lurt å se over kapitelet om generatorer for å få full forståelse av kodeeksemplene.

14.3 Innebygde funksjoner

Funksjonene som beskrives under, kan brukes på *iterables*.

La oss først repetere noen begreper:

Iterable

En *iterable* et objekt som kan returnere elementer ett om gangen, slik at det kan brukes i en løkke.

Sequence

En *sequence* (sekvens) er en type iterable *som har en bestemt rekkefølge og støtter indeksering*. Alle sekvenser er iterables, men ikke alle iterables er sekvenser. *Med tanke på bruk av funksjonene som er beskrevet har det ingen praktisk betydning om iterablen er en sekvens eller ikke, men det kan være greit å oppfriske forskjellen…*

Oversikt Python-iterabletyper

Type	Beskrivelse	Sekvens
`list`	Ordnet, endringsbar samling	Ja
`tuple`	Ordnet, uendringsbar samling	Ja
`str`	Ordnet sekvens av tegn	Ja
`range`	Ordnet sekvens av tall	Ja
`set`	Uordnet samling av unike elementer	Nei
`dict`	Samling av nøkkel-verdi-par	Nei
`frozenset`	Uordnet, uendringsbar versjon av set	Nei
`bytes`	Ordnet sekvens av byte-verdier	Ja
generator	Lazy iterable som produserer verdier én om gangen med `yield`	Nei

14.3.1 all()

`all()` returnerer en `bool`: `True` hvis **alle** elementene i en iterable er «truthy».

NB: Dersom iterablen er *tom* returneres også `True`.

Eksempel:

```
numbers = [2, 4, 6]
print(all(x % 2 == 0 for x in numbers)) # True

numbers = [2, 4, 6, 7]
print(all(x % 2 == 0 for x in numbers)) # False

numbers = []
print(all(x % 2 == 0 for x in numbers)) # True!!
```

Utskrift:

```
True
False
```

```
True
```

14.3.2 any()

`any()` returnerer en `bool`: `True` hvis *minst ett element* i en iterable er truthy.

NB: Dersom iterablen er *tom* returneres `False`!!.

Grunnen til at `all()` returnerer `True` på en tom mengde, mens `any()` returnerer `False`, bunner i logisk definisjon og matematisk konvensjon, uten at vi skal gå nærmere inn på det.

Eksempel:

```
numbers = [1, 3, 4]
print(any(x % 2 == 0 for x in numbers))  # True

numbers = [1, 3, 4, 6]
print(any(x % 2 == 0 for x in numbers))  # True

numbers = []
print(any(x % 2 == 0 for x in numbers))  # False!!
```

Utskrift:

```
True
True
False
```

14.3.3 map()

`map()` anvender en *funksjon* på alle elementene i en iterable.

`map()` kan ta *flere* iterables som input, vil stoppe når korteste iterable er brukt opp.

map() returnerer en *iterator*, nærmere bestemt et map object (som er en iterator)

Merk:
Mange vil hevde at comprehension er en mer pythonic måte å løse det `map()` gjør.

Eksempel:

```
numbers = [1, 2, 3]
squares = list(map(lambda x: x**2, numbers))
print(squares)  # [1, 4, 9]

numbers1 = [1, 2, 3, 4]
numbers2 = [10, 20, 30, 40]

# Funksjonen tar 2 parametere (x og y)
# Stopper når korteste list når slutten
result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))  # [11, 22, 33, 44]
```

Utskrift:

```
[1, 4, 9]
[11, 22, 33, 44]
```

Kommentar til koden:

Linja:

```
squares = list(map(lambda x: x**2, numbers))
```

Vi må bruke `list`-konstruktøren fordi resultatet av uttrykket `map(lambda x: x**2, numbers)` er et `map`-objekt, som er en iterator.

Et `map` iterator objekt er en iterator av type ***generator*** som beregner verdiene "lazy" - det vil si at verdiene ikke beregnes før de faktisk etterspørres. Dette gjør denne typen iteratorer minneeffektive, spesielt for store datasett.

Ved å bruke list-konstruktøren konsumerer vi iteratoren; - `list()` itererer gjennom `map`-objektet, henter alle beregnede verdier, og bygger opp en liste som kan brukes flere ganger.

Utrygg på lambda syntaks? Her er samme eksempel som over, uten lambda:

```
def square_it(x):
    return x**2

numbers = [1, 2, 3]
squares = list(map(square_it, numbers))
print(squares)  # [1, 4, 9]
```

14.3.4filter()

Filtrerer elementer basert på en betingelse.

```
numbers = [1, 2, 3, 4]
even_numbers = list(filter(lambda x: x % 2 == 0,
numbers))
print(even_numbers)  # [2, 4]
```

Utskrift:

```
[2, 4]
```

Merk:
Mange vil hevde at comprehension er en mer pythonic måte å løse det `filter()` gjør.

14.3.5zip() – kombinere flere iterables

`zip()` kombinerer flere iterables (lister, tupler, etc.) til en iterator av ***tupler***, hvor hvert element kommer fra samme posisjon i de opprinnelige iterablene.

Grunnleggende bruk:

```
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]

# Kombinerer to lister
for name, age in zip(names, ages):
    print(f"{name} er {age} år")
```

```
# Utskrift:
# Alice er 25 år
# Bob er 30 år
# Charlie er 35 år
```

Lage dictionary fra to lister:

```
keys = ['navn', 'alder', 'by']
values = ['Alice', 25, 'Oslo']

person = dict(zip(keys, values))
print(person)  # {'navn': 'Alice', 'alder': 25, 'by':
'Oslo'}
```

Flere enn to iterables:

```
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
cities = ['Oslo', 'Bergen', 'Trondheim']

for name, age, city in zip(names, ages, cities):
    print(f"{name} ({age}) bor i {city}")
```

Utskrift:

```
Alice (25) bor i Oslo
Bob (30) bor i Bergen
Charlie (35) bor i Trondheim
```

Ulik lengde, stopper ved korteste:

```
short = [1, 2]
long = [10, 20, 30, 40]

result = list(zip(short, long))
print(result)  # stopper etter 2 elementer
```

Utskrift:

[(1, 10), (2, 20)]

Under panseret; - `zip` er en iterator:

Som `map()` og `filter()`, `zip()` returnerer en iterator:

```
names = ['Alice', 'Bob']
ages = [25, 30]

zipped = zip(names, ages)
print(type(zipped))  # <class 'zip'>

# Må konvertere til liste for å se innholdet
print(list(zipped))  # [('Alice', 25), ('Bob', 30)]
```

Pakke ut med unpacking operatoren *:

```
pairs = [('Alice', 25), ('Bob', 30), ('Charlie', 35)]

# "Unzip" - pakk ut til separate lister
names, ages = zip(*pairs)
print(names)  # ('Alice', 'Bob', 'Charlie')
print(ages)   # (25, 30, 35)
```

14.3.6 `sorted()`: sortere iterables

Merk at **`sorted()`** og **`sort()`** er ganske forskjellige: **`sorted()`** ***returnerer en liste***, mens **`sort()`** er en metode som tilhører **`list`** klassen, og sorterer in-place, altså den sorterer lista på stedet. De er like på den måten at de kan ta som parameter en sorteringsfunksjon.

Grunnleggende bruk:

```
numbers = [3, 1, 4, 1, 5, 9, 2]
sorted_nums = sorted(numbers)
print(sorted_nums)  # [1, 1, 2, 3, 4, 5, 9]

# Original liste er uendret
print(numbers)  # [3, 1, 4, 1, 5, 9, 2]
```

Sortere i synkende rekkefølge:

```
numbers = [3, 1, 4, 1, 5]
desc = sorted(numbers, reverse=True)
print(desc)  # [5, 4, 3, 1, 1]
```

Sortere med key parameter:

key lar deg spesifisere en funksjon som skal brukes for å hente sorteringsnøkkel

```
words = ['banana', 'pie', 'Washington', 'book']

# Sortere etter lengde
by_length = sorted(words, key=len)
print(by_length)  # ['pie', 'book', 'banana',
'Washington']

# Sortere på siste bokstav
by_last = sorted(words, key=lambda w: w[-1])
print(by_last)  # ['banana', 'pie', 'book',
'Washington']
```

Sortere komplekse objekter:

```
students = [
    {'name': 'Alice', 'grade': 85},
    {'name': 'Bob', 'grade': 92},
    {'name': 'Charlie', 'grade': 78}
]

# Sortere etter karakter
by_grade = sorted(students, key=lambda s: s['grade'])
print([s['name'] for s in by_grade])  # ['Charlie',
'Alice', 'Bob']

# Sortere etter navn
by_name = sorted(students, key=lambda s: s['name'])
print([s['name'] for s in by_name])  # ['Alice', 'Bob',
'Charlie']
```

sorted() vs list.sort():

```
numbers = [3, 1, 4, 1, 5]
```

```
# sorted() - returnerer ny liste
new_list = sorted(numbers)
print(numbers)  # [3, 1, 4, 1, 5] (uendret)
print(new_list) # [1, 1, 3, 4, 5]

# list.sort() - endrer lista på stedet
numbers.sort()
print(numbers)  # [1, 1, 3, 4, 5] (endret)
```

14.3.7reversed() – reversere iterables

`reversed()` returnerer en iterator som går gjennom elementene i omvendt rekkefølge.

Grunnleggende bruk:

```
numbers = [1, 2, 3, 4, 5]
reversed_nums = list(reversed(numbers))
print(reversed_nums)  # [5, 4, 3, 2, 1]

# Original liste er uendret
print(numbers)  # [1, 2, 3, 4, 5]
```

`reversed()` er en iterator:

```
numbers = [1, 2, 3]
rev = reversed(numbers)
print(type(rev))  # <class 'list_reverseiterator'>

# Må konvertere til liste for å se innholdet
print(list(rev))  # [3, 2, 1]
```

Brukt i for-løkke:

```
for num in reversed([1, 2, 3, 4, 5]):
    print(num, end=" ")  # 5 4 3 2 1
```

Reversere strenger:

```
text = "Python"
```

```
reversed_text = ''.join(reversed(text))
print(reversed_text)  # nohtyP
```

reversed() vs slicing:

```
numbers = [1, 2, 3, 4, 5]

# Med reversed() - iterator (lazy)
rev1 = reversed(numbers)  # Iterator, ingen minne
brukt enda

# Med slicing - lager ny liste umiddelbart
rev2 = numbers[::-1]  # [5, 4, 3, 2, 1] (ny liste i
minnet)

# reversed() er mer minneeffektivt for store datasett
```

Under panseret: reversed():

`reversed()` er implementert i C som en egen klasse for ytelse. Den leser den opprinnelige sekvensen **baklengs** uten å lage en kopi.

`reversed()` er en **iterator** (ikke generator), men oppfører seg lazy som en generator. Dokumentasjonen kan si den yield'er elementer - det betyr bare at den produserer dem én om gangen, ikke at den bruker yield-keyword:

```
import types

r = reversed([1, 2, 3])

# Er det en iterator? JA
print(hasattr(r, '__iter__'))  # True
print(hasattr(r, '__next__'))  # True

# Er det en generator? NEI
print(isinstance(r, types.GeneratorType))  # False
print(type(r))  # <class 'reversed'> - egen klasse!
```

14.3.8 min(), max() sum() - aggregering

Disse funksjonene beregner minimum, maksimum og sum av en iterable.

14.3.8.1 min() og max():

```
numbers = [3, 1, 4, 1, 5, 9, 2]

print(min(numbers))  # 1
print(max(numbers))  # 9

# Fungerer også med strenger (leksikografisk)
words = ['banana', 'apple', 'cherry']
print(min(words))  # 'apple'
print(max(words))  # 'cherry'
```

Med key parameter:

```
words = ['banana', 'pie', 'Washington', 'book']

# Korteste ord
shortest = min(words, key=len)
print(shortest)  # 'pie'

# Lengste ord
longest = max(words, key=len)
print(longest)  # 'Washington'
```

Med komplekse objekter:

```
students = [
    {'name': 'Alice', 'grade': 85},
    {'name': 'Bob', 'grade': 92},
    {'name': 'Charlie', 'grade': 78}
]

# Student med laveste karakter
worst = min(students, key=lambda s: s['grade'])
print(worst['name'])  # 'Charlie'
```

```
# Student med høyeste karakter
best = max(students, key=lambda s: s['grade'])
print(best['name'])  # 'Bob'
```

14.3.8.2 sum()

```
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)
print(total)  # 15

# Med start-verdi
total_plus_10 = sum(numbers, 10)
print(total_plus_10)  # 25 (15 + 10)
```

Gjennomsnitt med sum() og len():

```
numbers = [10, 20, 30, 40, 50]
average = sum(numbers) / len(numbers)
print(average)  # 30.0
```

14.3.9 range() – generator for tallsekvenser

`range()` genererer en sekvens av tall. Det er en lazy generator som ikke lagrer alle tallene i minnet.

Grunnleggende syntaks:

```
# range(stop)
for i in range(5):
    print(i, end=" ")  # 0 1 2 3 4

# range(start, stop)
for i in range(2, 7):
    print(i, end=" ")  # 2 3 4 5 6

# range(start, stop, step)
for i in range(0, 10, 2):
    print(i, end=" ")  # 0 2 4 6 8
```

Negativ step (nedtelling):

```
for i in range(10, 0, -1):
    print(i, end=" ")  # 10 9 8 7 6 5 4 3 2 1
```

`range()` er en generator:

```
r = range(5)
print(type(r))  # <class 'range'>
print(r)        # range(0, 5)

# Må konvertere til liste for å se verdiene
print(list(r))  # [0, 1, 2, 3, 4]
```

Minneeffektivitet:

```
import sys

# range lagrer ikke tallene - tar minimal plass
r = range(1000000)
print(sys.getsizeof(r))  # ~48 bytes

# Liste lagrer alle tallene - tar mye plass
lst = list(range(1000000))
print(sys.getsizeof(lst))  # ~8 MB
```

Brukt med `enumerate`:

```
for i, num in enumerate(range(10, 15)):
    print(f"Index {i}: verdi {num}")
# Index 0: verdi 10
# Index 1: verdi 11
# ...
```

14.4Nyttige mekanismer

14.4.1Generatorer

I kapittel 10 så vi hvordan Iterator pattern lar oss gjennomløpe samlinger på en enhetlig måte. Vi lærte å implementere `__iter__()` og `__next__()` for å lage egne iteratorer. Nå skal vi se på generatorer - en kraftig mekanisme i Python som gjør det mye enklere å lage iteratorer.

14.4.1.1 Hva er en generator?

En generator er en spesiell type funksjon som genererer verdier én om gangen, i stedet for å returnere dem alle på en gang. Generatorer bruker nøkkelordet `yield` i stedet for return.

Enkel sammenligning:

```
# Vanlig funksjon - returnerer alt på en gang
def get_numbers():
    return [1, 2, 3]

# Generator-funksjon - genererer verdier én om gangen
def generate_numbers():
    yield 1
    yield 2
    yield 3
```

Iterator-eksempelet fra kapittel 10:

```
from collections.abc import Iterator

class MyIterator(Iterator):
    def __init__(self):
        self.data = [1, 2, 3]
        self.index = 0

    def __iter__(self):
        return self
```

```
    def __next__(self):
        if self.index < len(self.data):
            item = self.data[self.index]
            self.index += 1
            return item
        else:
            raise StopIteration

# Bruk:
my_iterator = MyIterator()
for item in my_iterator:
    print(item, end=" ")  # 1 2 3
```

Med generator blir dette:

```
def my_generator():
    data = [1, 2, 3]
    for item in data:
        yield item

# Bruk:
for item in my_generator():
    print(item, end=" ")  # 1 2 3
```

Fra 15 linjer til 4 linjer! Generator håndterer `__iter__()`, `__next__()`, tilstand og StopIteration automatisk.

14.4.1.2 Hvordan fungerer yield?

Når Python møter `yield` i en funksjon:

1. Funksjonen blir en **generator-funksjon**
2. Kall til funksjonen returnerer et **generator-objekt** (en iterator)
3. Hver gang `next()` kalles, kjører funksjonen til neste `yield`
4. Funksjonen "husker" hvor den var (tilstanden bevares automatisk)
5. Når funksjonen når slutten, kastes `StopIteration` automatisk

Eksempel:

```
def countdown(start):
    print("Starter nedtelling...")
    while start > 0:
        yield start  # "Pauser" her og returnerer
start
        start -= 1
    print("Ferdig!")

# Lager generator-objekt
gen = countdown(3)
print(type(gen))  # <class 'generator'>

# Henter verdier én om gangen
print(next(gen))  # Starter nedtelling... 3
print(next(gen))  # 2
print(next(gen))  # 1
print(next(gen))  # Ferdig! (StopIteration)
```

I kapittel10 implementerte vi også `__iter__()` i en egen klasse for å få en gjenbrukbar iterator:

```
class MyCollectionV2:
    def __init__(self, data):
        self._data = data

    def __iter__(self):
        return MyCollectionIterator(self._data)

class MyCollectionIterator:
    def __init__(self, data):
        self._data = data
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
```

```
        if self._index < len(self._data):
            item = self._data[self._index]
            self._index += 1
            return item
        else:
            raise StopIteration

# Bruk:
collection = MyCollectionV2([1, 2, 3])
for item in collection:
    print(item, end=" ")  # 1 2 3
for item in collection: # kan brukes pånytt…
    print(item, end=" ")  # 1 2 3
```

Vi kan omskrive dette med generator slik:

```
class MyCollectionV3:
    def __init__(self, data):
        self._data = data

    def __iter__(self):
        # Generator-funksjon erstatter
        # hele iterator-klassen!
        for item in self._data:
            yield item

# Bruk (identisk):
collection = MyCollectionV3([1, 2, 3])
for item in collection:
    print(item, end=" ")  # 1 2 3
for item in collection:
    print(item, end=" ")  # 1 2 3
```

Fra 25 linjer til 8 linjer! Og den er fortsatt gjenbrukbar fordi __iter__() lager en ny generator hver gang den kalles.

14.4.1.3 Generator expressions - kompakt syntaks

I tillegg til generator-funksjoner, finnes det generator expressions - en enda kortere syntaks som ligner list comprehensions:

```
# List comprehension - lager hele lista i minnet
squares_list = [x**2 for x in range(5)]
print(squares_list)  # [0, 1, 4, 9, 16]

# Generator expression - genererer verdier on-demand
squares_gen = (x**2 for x in range(5))
print(squares_gen)  # <generator object at 0x...>
print(list(squares_gen))  # [0, 1, 4, 9, 16]
```

Forskjell:

[...] - list comprehension - lager hele lista

(...) - generator expression - lazy evaluation

14.4.1.4 Fordeler med generatorer

Minneeffektive:

```
# Liste - bruker mye minne
big_list = [x**2 for x in range(1000000)]  # ~8 MB

# Generator - bruker nesten intet minne
big_gen = (x**2 for x in range(1000000))  # ~200 bytes
```

Enklere kode:

Sammenlign antall linjer:

- Iterator-klasse: 15-20 linjer
- Generator-funksjon: 3-5 linjer
- Generator expression: 1 linje

14.4.1.5 Praktiske eksempler:

Filtrering med generator:

```
def even_numbers(numbers):
    for n in numbers:
        if n % 2 == 0:
            yield n

# Bruk:
nums = [1, 2, 3, 4, 5, 6]
```

```
for even in even_numbers(nums):
    print(even, end=" ")  # 2 4 6
```

Fibonacci-sekvens:

```
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Bruk:
for num in fibonacci(10):
    print(num, end=" ")  # 0 1 1 2 3 5 8 13 21 34
```

14.4.1.6 Sammenligning Iterator vs Generator

Aspekt	**Iterator-klasse**	**Generator**
Kode	15-25 linjer	3-5 linjer
Tilstand	Manuell (`self._index`)	Automatisk
`__next__()`	Må implementere	Automatisk
`StopIteration`	Må reise manuelt	Automatisk
Kompleksitet	Høy	Lav
Når bruke	Kompleks tilstand, flere metoder	De fleste tilfeller

Bruk generator når:

- Enkel til moderat iterasjonslogikk
- Minneeffektivitet er viktig
- Du vil ha kortere, mer lesbar kode
- Du arbeider med store datasett eller filer

Bruk iterator-klasse når:

- Kompleks tilstandshåndtering
- Iterator trenger egne metoder (utover `__iter__` og `__next__`)
- Flere samtidige iteratorer med uavhengig tilstand

Bruk generator expression når:

- Enkel transformasjon/filtrering
- Én linje er tilstrekkelig
- Minneeffektivt alternativ til list comprehension

14.4.1.7 Generatorer og Iterator pattern

Generatorer er ikke en erstatning for Iterator pattern - de er en **enklere implementasjon** av det samme mønsteret:

- **Iterator pattern** definerer protokollen (`__iter__()` og `__next__()`)
- **Generatorer** implementerer denne protokollen ***automatisk***

```
# Generator implementerer Iterator pattern automatisk
def my_gen():
    yield 1
    yield 2

gen = my_gen()
print(hasattr(gen, '__iter__'))  # True
print(hasattr(gen, '__next__'))  # True
print(isinstance(gen, Iterator))  # True (fra collections.abc)
```

Kommentar til koden:

`hasattr()` er en innebygd Python-funksjon som sjekker om et objekt har en bestemt attributt (variabel eller metode).

14.4.1.8 Oppsummering generatorer

- **Generatorer** forenkler dramatisk opprettelsen av iteratorer
- `yield` erstatter manuell implementasjon av `__iter__()` og `__next__()`
- Python håndterer tilstand og `StopIteration` automatisk
- Generatorer er minneeffektive og kan representere uendelige sekvenser
- **Generator-funksjoner** (`def` med `yield`) for kompleks logikk
- **Generator expressions** (`(...)`) for enkle transformasjoner
- Generatorer implementerer Iterator pattern automatisk

Tommelfingerregel: Start med generatorer - bytt til iterator-klasser kun hvis du trenger kompleks tilstandshåndtering eller flere metoder.