

8 Klasser og objekter – grunnleggende

Læringsmål dette kapittel

[**<< kommer >>**](#)

8.1 Innledning

I Python og andre språk er klasser en måte å lage *egne* datatyper, der vi kan kombinere *data* (attributter) og *funktionalitet* (metoder) i én struktur. Et *objekt* er en *instans* av en klasse, altså en konkret forekomst av klassen.

Analogi til Python og det vi kan hittil:

```
number_1 = int(2)
number_2 = int(10)
```

number_1 og number_2 er objekter, int er klassen. Alle objekter laget på bakgrunn av int klassen har sine egne verdier (data / attributter), men oppfører seg likt.

Hittil har vi jobbet med innebygde klasser som int, str og list. Tenk på at int og list klassene er laget av Python programmererne. int er en relativt primitiv klasse, den kan lagre et heltall, returnere en strengrepresentasjon av seg selv, konvertere en streng til en int, etc. Med denne int klassen kan vi altså skape de konkrete objektene, enten slik

```
an_int = 1
eller
an_int = int(1)
```

I det første tilfellet skjønner Python oversetteren at det er int objekt vi ønsker å skape, gitt konteksten.

I det andre tilfellet bruker vi int konstruktøren for å lage int objektet på venstre side.



Teknikken ovenfor, `an_int = 1` kan kun brukes for visse innebygde datatyper. For våre egne klasser må vi bruke teknikken som tilsvarer

`an_int = int(1)`, altså vi bruker klassens *konstruktør* (det vil si `__init__()` metoden som vi ser nærmere på nedenfor) for å lage et objekt av klassen.

`list` klassen er mye mer avansert, den kan lagre en samling av data, sortere de, returnere en streng representasjon av seg selv, den kan itereres gjennom, samt at den kan masse annet. På samme måte som de innebygde klassene er enten enkel eller avansert, på samme måte kan våre klasser være enkel eller avansert.

Så, nå skal vi lære hvordan vi kan lage våre *egne* klasser. Vi begynner med enkle, konseptuelle klasser hvor vi forklarer det grunnleggende, og blir mer avanserte etter hvert.

8.2 Lage en klasse

En klasse kan sees på som en *mal* eller *oppskrift* for å lage objekter. For eksempel kan vi lage en klasse `Car` som beskriver egenskaper (som farge og modell) og oppførsel (som å kjøre eller stoppe) til en bil. Hvert objekt vi lager fra klassen, for eksempel en spesifikk bil, vil ha *sine egne* verdier for disse egenskapene. Hvert bilobjekt, som en blå Toyota eller en rød Volvo, er en *instans* av `Car`-klassen med unike verdier.

```
# file sc_08_01.py
1. class Car:
2.     def __init__(self, brand, color):
3.         self._brand = brand
4.         self._color = color
5.
6.     def drive(self):
7.         print(f"{self._brand} car is driving!")
8.
9.     def show_info(self):
10.        print(f"Car: {self._brand}, Color: {self._color}")
11.
12. # Create two car objects
13. my_car = Car("Toyota", "Blue")
14. another_car = Car("Volvo", "Red")
15.
16. # Use the methods
17. my_car.drive()
18. my_car.show_info()
19. another_car.show_info()
```

Utskrift (fra linje 17, 18, og 19):

```
Toyota car is driving!
Car: Toyota, Color: Blue
Car: Volvo, Color: Red
```

Forklaring til koden:

Linje 1

Definerer Car-klassen. Vi bruker Python nøkkelordet **class** for å definere en klasse. Klassenavn i Python bør begynne med en *stor* bokstav, hvis flere ord; - stor bokstav begynner hvert ord.

Linje 2-4

`_init_()` - metoden deklarerется og initialiseres attributtene `_brand` og `_color` med verdiene som sendes inn. *Enkel understrek* indikerer at attributtene er *for intern bruk*, hvilket betyr at kode utenfor klassen ikke skal modifisere disse.

Se linje 13 og 14; - slik skaper vi et objekt, og argumentene i parentesene etter klassenavnet er det som bli parametriene til `_init_()` i tillegg til den først parameteren, `self`. Vi forklarer `self` inngående om ei stund...

Linje 6-7

Definerer `drive()`-metoden, som simulerer at bilen kjører. Objektets `_brand`-attributt skrives ut.

Linje 9-10

Definerer `show_info()`-metoden, som skriver ut bilens merke og farge.

Linje 13-14

Oppretter referanser til to Car-objekter, `my_car` og `another_car`, med spesifikke verdier for merke og farge.

Linje 17-19

Kaller metodene `drive()` og `show_info()` på objektene for å demonstrere deres atferd.

8.3 `__init__()` metoden

`__init__()` - metoden, også kalt *konstruktøren*, er en spesiell metode som kalles **automatisk** (implisitt) når et objekt opprettes (linje 13 og 14). Den *initialiserer objektets attributter*. Parameteren `self` refererer til instansen (objektet) som opprettes, og ytterligere parametere kan ta imot data for å sette opp objektet. For eksempel, i `Car`-klassen, bruker `__init__()` parametrene `brand` og `color` for å definere bilens egenskaper, som er attributtene `_brand` og `_color`.

8.4 Hva er `self`?

Det er viktig å forstå følgende:

Koden som er definert i en klasse **deles av alle objektene**.

Attributtene, altså dataene som er definert, **er unike for hvert objekt som opprettes**.

I klassen over er attributtene til klassen `_brand` og `_color`. I linje 13 og 14 oppretter vi to `Car` objekter ved hjelp av `__init__()` metoden, men legg merke til at mens `__init__()` metoden skal ha tre parametere; - `self`, `brand` og `color`, så er `self` fraværende i metodekallet i disse linjene. Det samme i linjene 17 til 19, her påkaller vi metodene `drive()` og `show_info()`, men vi sender ikke inn `self` parameteren selv om metodesignaturen sier at vi skal ha en slik. Så hvorfor godtar Python interpreteren at det «mangler» en parameter, nemlig `self`, til disse metodene?

Svaret er at Python (som så mange andre steder) gjør noe bak kulissene. Kallet
`my_car.drive()`

skrives *om* av Python til
`Car.drive(my_car)`

Altså vi påkaller klassen `Car` sin `drive()` metode ved hjelp av *dot notasjonen*.

Det siste kallet stemmer med metodesignaturen til `drive()` metoden som er
`def drive(self)`

Det samme er tilfellet når vi påkaller `_init_()`, kallet
`my_car = Car("Toyota", "Blue")`
blir omskrevet til
`Car.__init__(my_car, "Toyota", "Blue")`

Den oppmerksomme leser vil se at det er noe som skurrer i koden:
`Car.__init__(my_car, "Toyota", "Blue")`

Hvordan kan vi sende `my_car` som første parameter, når den ennå ikke er opprettet?

En slags «høna og egget» problematikk...

Svaret er at konstruksjonen av et objekt i Python skjer i **to trinn**:

1. Først kalles en metode `__new__()`, som **allokerer minne og oppretter selve objektet**.
2. Deretter kalles `__init__()`, som **initialiserer objektets tilstand** (f.eks. setter attributter).
3. Det betyr at når `__init__()` blir kalt, **eksisterer objektet allerede**, og det er derfor vi kan sende det inn som første argument (`self`).

Uansett vet du nå at hver instansmetode i `Car` klassen faktisk får en referanse til det objektet metoden skal jobbe på, og dermed de rette attributtene, de som tilhører akkurat dette objektet.

I tillegg til instansmetoder har vi *klassemetoder*, som evt jobber på data som tilhører klassen, ikke de enkelte objektene. Mer om det senere.

8.5 . dot notasjonen

dot notasjonen forteller Python at vi ønsker å bruke *noe som tilhører objektet* – enten en verdi (attributt) eller en funksjon (metode). Uten punktum vet ikke Python hvilket objekt som menes, og du får en feilmelding.

Hva skjer egentlig?

Når du skriver `my_car.drive()`, så ber du objektet `my_car` om å kjøre koden for metoden `drive()`, anvendt på den egne attributten

`_brand`. Som vi har sett tilhører koden *klassen*, så `my_car` referansen er nødvendig for å fortelle hvilket objekt koden i `drive()` skal kjøres med hensyn på.

8.6 Hva er det som kan / bør bli en klasse?

Når du skal lage et program vil du ofte møte på situasjoner der du må modellere noe – en ting, en person, en prosess, en struktur – som har både **data** og **oppførsel**. Da bør du spørre deg selv: *Er dette noe som kan bli en klasse?*

Når bør du lage en klasse?

Du bør vurdere å lage en klasse når:

- Du har **flere relaterte data** som hører sammen
- Du trenger **funksjoner** som opererer på disse dataene
- Du ønsker å **organisere** koden på en ryddig og gjenbrukbar måte
- Du skal lage **flere instanser** (objekter) med samme struktur, men ulikt innhold

Eksempler:

- En Student med navn, ID, karakterer og metoder for å beregne snitt.
- En Account med saldo, rente og metoder for innskudd og uttak.
- En Book med tittel, forfatter, ISBN og metoder for å vise info eller sjekke tilgjengelighet.

Hva bør ikke bli en klasse?

Ikke alt trenger å bli en klasse. Hvis du bare har én verdi eller en enkel funksjon, er det ofte nok med en vanlig variabel eller funksjon.

Eksempel:

```
def calculate_area(radius):
    return 3.14 * radius ** 2
```

Her er det ikke behov for en klasse – funksjonen gjør én ting, og det er greit.

Et godt spørsmål å stille:

Har det jeg prøver å modellere både data og oppførsel som hører sammen?

Hvis svaret er ja er det sannsynligvis en god kandidat for en klasse.

8.7 Hvor skal klassens attributter opprettes / initieres?

Attributter (datamedlemmer) kan i prinsippet opprettes i en hvilken som helst av klassens metoder.

De bør / skal opprettes i `__init__()` metoden:

- **Oversikt og forutsigbarhet:** Alle attributter som tilhører objektet blir tydelig definert når objektet lages.
- **Unngå bugs:** Hvis attributter opprettes tilfeldig i ulike metoder, kan det føre til at noen metoder prøver å bruke attributter som ikke er satt ennå.

Normalt har data opprettet i en metode *lokalt scope*...

Legg merke til at attributtene opprettes med `self._navn`.

Det betyr at de opprettes *på objektet*, og lever så lenge objektet eksisterer.

8.8 En mer avansert klasse: Account

La oss se på en litt mer avansert klasse, `Account`, som skal representere en konto i en bank. Vi skal utvide klassen i flere trinn for å vise aspekter både rent «teknisk» rundt en klasse, samt design av en løsning som innebærer samarbeid / interaksjon mellom flere klasser.

Vi skal se på følgende utgaver av kontosystemet vårt:

1. En selvstendig konto klasse
2. Konto klasse med transaksjoner (egen transaksjonsklasse)
3. Konto klassen med transaksjoner, egen transaksjonsklasse og *properties*

8.8.1 Kun Account klassen

Kode for første variant av Account klassen:

```

1 # file: sc_08_Account1.py
2 class Account:
3     def __init__(self, cust_id, account_no, start_balance, interest):
4         self._cust_id = cust_id
5         self._account_no = account_no
6         self._balance = start_balance
7         self._interest = interest
8
9     def get_balance(self):
10        return self._balance
11
12    def set_balance(self, new_balance):
13        self._balance = new_balance
14
15    def deposit(self, amount):
16        if amount > 0:
17            self._balance += amount
18        return self._balance
19
20    def withdraw(self, amount):
21        if amount <= self._balance:
22            self._balance -= amount
23        return self._balance
24
25    def add_monthly_interest(self):
26        monthly_interest = self.calculate_monthly_interest()
27        self._balance += monthly_interest
28
29    def calculate_monthly_interest(self):
30        return self._balance * self._interest / 100 / 12
31
32    def __str__(self):
33        return f'''
34 Customer id   = {self._cust_id}
35 Account no    = {self._account_no}
36 Balance       = {self._balance}
37 Interest      = {self._interest}
38 '''

```

Forklaring til koden:

Linje 2

Definerer Account som en Python klasse

Linje 3 – 7

`__init__()` metoden tar imot kundeidentifikasjon (`cust_id`), kontonummer (`account_no`), startbeløp/saldo (`start_balance`) og rente (`interest`). Merk at `interest` oppgis som et *heltall*; - dersom

renten er 5%, vil `interest` være 5. Merk at i en renteberegning i forhold til en saldo må vi bruke 0.05 når vi multipliserer for å finne renten...

Disse verdiene tilordnes dataattributtene til klassen. Legg merke til at alle attributtene har en innledende `_` (underscore) i navnet; - dette er en konvensjon i Python som forteller at klientkode ***ikke*** skal endre verdien på attributten, direkte; - en endring skal eventuelt skje på andre måter. Vi vil likevel se at vi lett kan endre de «private» attributtene fra klientkode.

Linje 9 – 13

Metoden `get_balance()` brukes for å hente ut saldo, mens `set_balance()` brukes til å sette en ny saldo. Poenget er at alle attributtene er markert «privat», med en innledende `_`, og vi skal bruke andre teknikker for å lese / skrive til de enn å bruke attributtnavnet direkte.

Linje 15 – 18

Metoden `deposit()` brukes for innskudd. Ny saldo returneres.

Linje 20– 23

Metoden `withdraw()` brukes for uttak, og det gjøres en test på at ikke saldo blir negativ. Ny saldo returneres.

Linje 25– 30

Metodene `add_monthly_interest()` og `calculate_monthly_interest()` defineres. Dette er metoder for å beregne månedlig rente, samt legge dette til saldo.

På linje 30 deles det på 100 og 12. Se forklaring for 100-delingsa på linje 3 – 7, og husk vi skal ha månedlig rente, derfor deles det i tillegg på 12...

Linje 32 – 38

Metoden `__str__()` er en av metodene som finnes i `object` klassen, den klassen som er basis for alle klasser i Python, også vår `Account` klasse. `__str__()` bruker her en *f-string* med triple quotes for å lage en streng som går over flere linjer.

Se egen forklaringsboks om `__str__()` metoden nedenfor, men kort fortalt kalles `__str__()` metoden *implisitt* når vi f eks oppgir en referanse til et `Account` objekt som parameter til `print()` metoden.

Se også forklaring til kode nedenfor (klientkoden som bruker `Account` klassen)

Klientkode for `Account` klassen

Vi trenger noe kode for å teste `Account` klassen vår. Klientkode er et begrep som brukes om kode som *bruker / utnytter* annen kode.

```

1 # file: sc_08_useAccount1.py
2 from sc_08_Account1 import Account
3
4 account1 = Account(1, 1000, 50000, 7)
5 account2 = Account(2, 1001, 10000, 5)
6 print(account1)
7 print(account2)
8 account1.deposit(500)
9 # Account.deposit(account1, 500)
10 print(f'New balance after depositing: {account1.get_balance()}')
11 account1.withdraw(1000)
12 print(f'New balance after withdrawal: {account1.get_balance()}')
13 account1.add_monthly_interest()
14 print(f'New balance after monthly interest: {account1._balance}')

```

Forklaring til kode:

Linje 2

Importerer `Account` fra modulen `sc_08_Account1.py`.

Linje 4 og 5

Oppretter to objekter fra `Account` klassen. Her kalles `__init__()` i `Account` klassen med nødvendige parametere. Husk at en referanse (`self`) til `account1` / `account2` objektene blir satt inn av Python oversetteren som første parameter i kallene, ref tidligere forklaring

Linje 6 og 7

Her skriver vi ut objektene. Når en referanse (f eks `account1` og `account2`) er input til `print()` så vil Python oversetteren lete etter en metode i `Account` klassen som heter `__str__()`. Hvis metoden finnes, så kalles den. Hvis ikke kalles `__str__()` metoden som ligger i `object` klassen.

Linje 8

Her setter vi inn 500 ved hjelp av `deposit()` metoden

Linje 9 (kommentert ut)

Bare for å demonstrere at det finnes en alternativ måte å kalle `deposit()` på, ref tidigere forklaring av `self` parameteren. Prøv å kjøre denne linja i stedet for linje 8, det skal ikke være noen forskjell.

Linje 10 – 14

Skriver ut saldo etter innskudd, linje 8 eller 9. Utfører uttak på linje 11, og legger til månedlig rente på linje 13. På linje 14 vises det at vi kan aksessere attributten `_balance` direkte, vi behøver altså ikke å bruke `get_balance`. I motsetning til de fleste andre språk har ikke Python en mekanisme som gjør at du kan beskytte attributtene mot direkte aksess; - det at du putter en `_` (underscore) foran et attributtnavn er kun et signal til brukere av vår `Account` klasse om at denne attributten skal du ikke aksessere direkte.

Utskriften fra programmet blir:

```
Customer id = 1
Account no = 1000
Balance = 50000
Interest = 7
```

```
Customer id = 2
Account no = 1001
Balance = 10000
Interest = 5
```

```
New balance after depositing: 50500
New balance after withdrawal: 49500
New balance after monthly interest: 49788.75
```

8.9 Synlighet av datamedlemmer

Det er en ting som spesielt overrasker programmerere som kommer fra C++ / Java miljøer: Python har ikke mulighet for å deklarerere at datamedlemmer eller metoder skal være `private`.

private?

I C++ / Java, hvis du deklarerer at et datamedlem skal være `private`, så betyr det at kun kode skrevet innenfor selve klassen kan aksessere disse datamedlemmene. Kode utafor klassen, klientkode, kan ikke

aksessere datamedlemmet. Denne mekanismen bidrar til de vi kaller *innkapsling*, som er et konsept for å beskytte klassen mot utilsiktet endring fra kode utenfor klassen.

I Python er *alle datamedlemmer og metoder public*, det vil si at datamedlemmene / metodene er tilgjengelig fra klientkode.

Merk at `public` er et keyword som ikke finnes i Python, så ordene `public` og `protected` nevnes kun fordi vi gjør sammenligningen med andre språk.

Så, gitt vår første versjon av `Account` klassen, så kan vi utføre følgende fra klientkode:

```
from sc_08_Account1 import Account
konto = Account("Ola", 1000)
print(konto._name)      # F.eks. "Ola"
print(konto._balance)   # F.eks. 1000

# Endre attributtene direkte fra klientkode:
konto._name = "Kari"
konto._balance = 5000

print(konto._name)      # Nå "Kari"
print(konto._balance)   # Nå 5000
```

Vi ser at vi uten problem kan endre attributtene `_name` og `_balance` fra klientkode.

8.9.1 Double underscore («dunder») for å gjøre datamedlemmer private?

I Python kan du gjøre attributter "private" ved å starte navnet med dobbel underscore, for eksempel `self._balance`. Dette kalles *name mangling*, og gjør at attributtet ikke er *direkte* tilgjengelig fra utsiden. Men det er fortsatt mulig å nå attributtet hvis man kjenner navnet (f.eks. `_Account__balance`).

Eksempel på bruk av double underscore:

```
class Account:
```

```

def __init__(self, name, balance):
    self.__name = name
    self.__balance = balance

konto = Account("Ola", 1000)
# konto.__balance # Gir AttributeError
print(konto._Account__balance) # tilgang likevel..

```

8.9.2 Name mangling?

Name mangling er en mekanisme i Python der attributter som starter med to understreker (f.eks. `__balance`) *automatisk* får navnet endret til `_Klassenavn__balance` internt.

Hensikt:

- Name mangling ble laget for å unngå navnekonflikter i *arv* (subklasser).
- Hvis både en superklasse og en subklasse har et attributt med samme navn (og dobbel underscore), vil de ikke overskrive hverandre, fordi Python endrer navnet basert på klassen.

Name mangling ble altså ikke funnet opp fordi en skulle lage en mekanisme for å lage private datamedlemmer, og det anbefales ikke å bruke name mangling til dette formålet.

8.9.3 «Private» datamedlemmer

Python har valgt å ikke ha “ekte” private datamedlemmer slik som C++ og Java, fordi språket bygger på prinsippet om “tillitt mellom utviklere” (the consent of adults). I stedet for å tvinge fram strenge regler, stoler Python på at programmereren følger konvensjoner (f.eks. én underscore for “internt bruk”). Dette gjør koden mer fleksibel, enklere å lese og lettere å teste, og gir utvikleren frihet til å gjøre bevisste valg – heller enn å bli stoppet av språket.

Det er en konvensjon i Python at dersom du ønsker å si «klientkode, legg unna dette datamedlemmet», så prefikser du datamedlemmet med en underscore. Dette er Python sin måte å si at dette datamedlemmet er privat.

8.9.4 Account og transactions

Den neste utgaven av `Account` introduserer en egen klasse for transaksjoner slik at vi får logget hva som skjer.

Kode for `Transaction` og `Account` ligger i samme .py fil:

```
# file: sc_08_02_account2.py
from datetime import datetime

class Transaction:
    def __init__(self, amount, trans_type):
        self._amount = amount
        self._trans_type = trans_type # "deposit", "withdraw",
"interest"
        self._timestamp = datetime.now()

    def __str__(self):
        return f"{self._timestamp:%Y-%m-%d %H:%M:%S} | {self._trans_type.capitalize():8} | {self._amount:8.2f}"
```

Forklaring til koden (Transaction klassen):

`Transaction` klassen har attributter som registrerer beløp, transaksjonstype og en timestamp.

Det lages transaksjoner når `Account` klassen sine metoder `withdraw()`, `deposit()` og `add_monthly_interest()` sine metoder utføres, og `__str__` metoden kalles når `Account` metoden `print_transaction()` utføres (implisitt kall pga `print` setningen).

Det er `Account` klassen sitt ansvar å lagre `Transaction` objektene i en liste `_transactions` i `Account` klassen.

```
class Account:
    def __init__(self, cust_id, account_no, start_balance,
interest):
        self._cust_id = cust_id
        self._account_no = account_no
        self._balance = start_balance
        self._interest = interest
        self._transactions = []

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
```

```

        self._transactions.append(Transaction(amount,
"deposit"))
        return self._balance

    def withdraw(self, amount):
        if amount <= self._balance:
            self._balance -= amount
            self._transactions.append(Transaction(amount,
"withdraw"))
        return self._balance

    def add_monthly_interest(self):
        monthly_interest = self.calculate_monthly_interest()
        self._balance += monthly_interest
        self._transactions.append(Transaction(monthly_interest,
"interest"))

    def calculate_monthly_interest(self):
        return self._balance * self._interest / 100 / 12

    def get_transactions(self):
        return self._transactions

    def print_transactions(self):
        print("Dato og tid | Type | Beløp")
        print("-" * 42)
        for trans in self._transactions:
            print(trans)

    def __str__(self):
        return f'''
Customer id  = {self._cust_id}
Account no   = {self._account_no}
Balance      = {self._balance:.2f}
Interest     = {self._interest}%
'''
```

Forklaring til koden (Account klassen)

__init__ metoden:

Her introduseres en liste

`self._transactions = []`

Denne brukes til å lagre Transaction objekter, etterhvert som de skapes fra metodene `withdraw()`, `deposit()` og `add_monthly_interest()`.

**`withdraw()`,
`deposit()` og
`add_monthly_interest()` metodene:**

Disse lager Transaction objekter og setter de inn i lista `_transactions`.

`print_transactions()` metoden:

Denne lager en pen utskrift av transaksjonene til kontoen.

Klientkode som bruker vår nye konto klasse:

```
from sc_08_Account2 import Account
account1 = Account(1, 1000, 50000, 7)

# Gjør innskudd og uttak
account1.deposit(200)
account1.withdraw(100)

# Skriv ut transaksjoner
def print_transactions(account):
    print("Dato og tid | Type | Beløp")
    print("-" * 42)
    for trans in account._transactions:
        print(trans)

print_transactions(account1)

# Skriv ut account1info
print(account1)
```

Utskrift:

Dato og tid	Type	Beløp
<hr/>		
2025-09-23 10:10:52	Deposit	200.00
2025-09-23 10:10:52	Withdraw	100.00

Customer id = 1

Account no	= 1000
Balance	= 600.00
Interest	= 3.5%

8.9.5 Properties - innledningsvis

Properties er den beste (og mest pythoniske) måten å beskytte attributter på i Python. Med properties kan du kontrollere lesing og skriving, for eksempel legge til validering eller gjøre attributter skrivebeskyttet.

Når du bruker properties, så aksesserer du attributtene uten den innledende underscore, så attributtnavnet for `_balance` blir `balance`, ikke `_balance`.

Properties bruker en såkalt *dekoratør*, og en god del ting skjer «under panseret», ting vi kommer tilbake til i senere kapitler. Det som er viktig nå er hvordan vi kan bruke properties til å kontrollere les- og skriveaksess til attributter.

Dekoratør

En dekoratør (decorator) i Python er en funksjon som endrer eller utvider funksjonaliteten til en annen funksjon eller metode, uten å endre selve koden i funksjonen. Dekoratører skrives med @-tegn over funksjonen/metoden, for eksempel `@property`, `@staticmethod`, eller `@classmethod`. De brukes for å legge til ekstra oppførsel på en enkel og lesbar måte.

Merk: *Selv med properties er ikke attributtene «ekte private».*
Python har ingen ekte private attributter – alt kan nås hvis man virkelig vil. *Properties gir deg kontroll og fleksibilitet, men bygger fortsatt på tillit mellom utviklere.*

8.9.6 Account, Transactions og properties

Neste utgave av Account introduserer *properties*.

Transaction klassen er uforandret og tas ikke med. Metodene fra `deposit()` og nedover (ref forrige variant av Account, se over) er heller ikke endret og tas ikke med.

```
# file: sc_08_02_account3.py
class Account:
    def __init__(self, cust_id, account_no, start_balance,
interest):
        self._cust_id = cust_id
        self._account_no = account_no
        self._balance = start_balance
        self._interest = interest
        self._transactions = []

@property
def balance(self):
    return self._balance

@balance.setter
def balance(self, new_balance):
    if new_balance < 0:
        print("Saldo kan ikke være negativ!")
    else:
        self._balance = new_balance

@property
def interest(self):
    return self._interest

@interest.setter
def interest(self, new_interest):
    if new_interest < 0:
        print("Rente kan ikke være negativ!")
    else:
        self._interest = new_interest
# resten av koden tas ikke med, uendret fra forrige versjon
```

Metodene prefikset med @property, f eks

```
@property
def balance(self):
    return self._balance
```

... er *getter*-metoder, mens metodene som er prefikset med
@xxxx.setter, f eks

```
@balance.setter
def balance(self, new_balance):
    if new_balance < 0:
        print("Saldo kan ikke være negativ!")
```

```
else:  
    self._balance = new_balance
```

... er *setter*- metoder.

Legg merke til at setter-metoden for balance har logikk innebygd, vi får ikke satt saldo til en negativ verdi. Dette er en av de store gevinstene ved properties.

Klientkode kan bruke disse på flg måte:

```
print(f"\nSaldo etter transaksjoner: {account1.balance:.2f}")  
account1.balance = 2000
```

Legg merke til:

Begge kodelinjene aksesserer attributten på samme måte:

`account1.balance`,
forskjellen er at i det ene tilfellet *leser* vi attributten, og i det andre tilfellet *skriver* vi til attributten.

Her, i klientkoden er attributtnavnet `balance`, ikke `_balance`, i property metodene brukes `_balance`.

For å lage en lese- og skrive-property til en attributt i Python følger du denne oppskriften:

Selve attributten må følge konvensjonen for et «privat» datamedlem, altså *en ledende _ (underscore)*.

Lag en metode med `@property`-dekoratøren:

Metoden skal ha navnet til attributten, *uten* den ledende underscore. Denne metoden brukes for å lese verdien (`get`), og den *kalles implisitt når attributtnavnet står på høyre side av "=" i klientkode*.

Lag en metode med samme navn, men med `@<navn>.setter`.

Metoden skal ha navnet til attributten, *uten* den ledende underscore. Denne metoden brukes for å sette (endre) verdien, og den *kalles implisitt når attributtnavnet står på venstre side av "=" i klientkode*.

8.9.7 Tips for når du bør bruke properties i Python

Når du vil kontrollere eller validere endringer:

Bruk property hvis du vil sjekke eller begrense hva som kan settes (f.eks. saldo kan ikke være negativ).

Når du vil skjule intern implementasjon:

Hvis du senere vil endre hvordan en verdi lagres eller beregnes, kan property gi samme grensesnitt utad.

Når du vil gjøre et attributt skrivebeskyttet:

Du kan lage en property uten setter, slik at verdien bare kan leses.

Når du vil utføre beregninger “på farten”:

Bruk property for å returnere verdier som regnes ut basert på andre attributter.

Når du vil logge eller spore tilgang:

Property kan brukes til å logge hver gang en verdi leses eller endres.

Ikke bruk property hvis:

Attributtet bare skal lagres og hentes uten ekstra logikk.
Klassen kun er en enkel databeholder (f.eks. Transaction).

Oppsummert:

Bruk property når du trenger kontroll, validering, fleksibilitet eller skjuling av intern logikk – ellers er vanlige attributter enklest!

8.9.8 Get metode eller property getter?

I den første utgaven av Account klassen hadde vi en `get_balance()` metode som vi erstattet med en property.

Det anbefales å bruke en property-getter (med `@property`) når attributtet skal oppføre seg som en vanlig variabel, altså når det er naturlig å skrive `objekt.verdi` i stedet for `objekt.get_verdi()`. Dette gir mer lesbar og «pythonsk» kode, og skjuler om verdien faktisk beregnes eller hentes fra et felt.

V kan bruke en get-metode (f.eks. `get_verdi()`) hvis:

- Du ønsker å signalisere at det skjer noe spesielt eller potensielt kostbart (f.eks. databaseoppslag, nettverkskall).
- Du vil at det skal være tydelig for brukeren at det er en funksjon, ikke et felt.
- Du følger et API eller en stil der get/set-metoder er standard (f.eks. i noen rammeverk eller ved interoperabilitet med Java/C++-kode).

Et eksempel der en get-metode kan være mer passende enn en property, er hvis du skal hente alle transaksjoner for en konto fra en database eller en ekstern tjeneste, altså en operasjon som kan ta tid eller kreve ekstra ressurser:

```
def get_transactions_from_db(self):
    # Henter transaksjoner fra database (kan ta tid)
    # ... kode for databaseoppslag ...
    return transactions
```

8.10 Klassemetoder og klassevariabler

De datamedlemmene / attributtene vi har sett på hittil tilhører *objektene*.

Det vil også være behov for data som tilhører *klassen*, som da vil være en felles ressurs for alle objektene. Eksempler for vår Account klasse kan være

- neste ledige kontonummer
- antall kontoer opprettet
- antall transaksjoner utført

Disse variablene opprettes inne i klassen, gjerne helt først etter klassedeklarasjonen, og utenfor metoder.

Klassemetodene kan være en av to typer: *class method* og *static method*. Forskjellen på disse er:

Class method

- Brukes for å lese eller endre klassevariabler
- Får klassen (cls) som første argument, ikke objektet (self)
- Må annoteres med @classmethod for at Python skal vite at metoden skal motta klassen som første parameter
- Ikke vanlig å implementere disse som properties (og heller ikke «rett fram» å få de til å bli properties)

Eksempel:

```
class A:
    count = 0
    @classmethod
    def increment(cls):
        cls.count += 1
```

Static method

- Metoden får verken self (objektet) eller cls (klassen).
- Kan ikke endre objekt- eller klassevariabler direkte.
- Brukes for funksjoner som logisk hører til klassen, men ikke trenger tilgang til objekt eller klasse.

Eksempel:

```
class A:
    @staticmethod
    def add(x, y):
        return x + y
```

Altså:

- Bruk @classmethod når du trenger tilgang til klassen.
- Bruk @staticmethod for hjelpefunksjoner som ikke trenger tilgang til verken objekt eller klasse.

I det neste eksempelet, som kun berører Account klassen, har vi lagt til tre klassevariabler _next_account_no, _account_count og _transaction_count, og

- en static method som sjekker at et kontonummer er ok
- en class method som får tak i verdien til klassevariabelen _account_count

- en class method som får tak i verdien til klassevariabelen `_transaction_count`
- en class method som setter ny startverdi i klassevariabelen `_next_account_no`

Kode for ny Account klasse; - de metodene som ikke er berørt av klassevariablene og klassemетодene som er innført, er ikke tatt med:

```

1: # file: sc_08_Account4.py
2: from datetime import datetime
3:
4: class Account:
5:     @staticmethod
6:         def is_valid_account_no(account_no):
7:             # Returnerer True hvis int og innenfor intervall
8:             return isinstance(account_no, int) and 1000 <=
account_no <= 9999
9:             _next_account_no = 1000    # Klassevariabel
10:            _account_count = 0        # Klassevariabel
11:            _transaction_count = 0   # Klassevariabel
12:
13:        @classmethod
14:            def set_next_account_no(cls, new_start):
15:                if new_start > cls._next_account_no:
16:                    cls._next_account_no = new_start
17:
18:        @classmethod
19:            def get_account_count(cls):
20:                return cls._account_count
21:
22:        @classmethod
23:            def get_transaction_count(cls):
24:                return cls._transaction_count
25:
26:        def __init__(self, cust_id, start_balance, interest):
27:            self._cust_id = cust_id
28:            self._account_no = Account._next_account_no
29:            Account._next_account_no += 1
30:            Account._account_count += 1
31:            self._balance = start_balance
32:            self._interest = interest
33:            self._transactions = []
34:
35:
36:        def deposit(self, amount):

```

```

37:         if amount > 0:
38:             self._balance += amount
39:             self._transactions.append(Transaction(amount,
"deposit"))
40:             Account._transaction_count += 1
41:         return self._balance
42:
43:     def withdraw(self, amount):
44:         if amount <= self._balance:
45:             self._balance -= amount
46:             self._transactions.append(Transaction(amount,
"withdraw"))
47:             Account._transaction_count += 1
48:         return self._balance
49:
50:     def add_monthly_interest(self):
51:         monthly_interest = self.calculate_monthly_interest()
52:         self._balance += monthly_interest
53:
54:         self._transactions.append(Transaction(monthly_interest,
"interest"))
55:
56:
57:
58: # Eksempel på bruk:
59: if __name__ == "__main__":
60:     # Sett nytt startpunkt for neste ledige kontonummer
61:     Account.set_next_account_no(2000)
62:
63:     acc1 = Account("A123", 1000, 2.5)
64:     acc2 = Account("B456", 500, 1.8)
65:     acc1.deposit(200)
66:     acc2.withdraw(100)
67:     acc1.add_monthly_interest()
68:
69:     print(acc1)
70:     print(acc2)
71:     print(f"Antall kontoer: {Account.get_account_count()}")
72:     print(f"Antall transaksjoner:
{Account.get_transaction_count()}")
73:
74:     # Eksempel på bruk av staticmethod:
75:     test_no = 2345
76:     if Account.is_valid_account_no(test_no):
77:         print(f"{test_no} er et gyldig kontonummer.")
78:     else:
79:         print(f"{test_no} er IKKE et gyldig kontonummer.")

```

Kommentarer til koden:**Linje 5 – 8**

Her defineres en statisk klasse metode som sjekker om et kontonummer er gyldig. `isinstance` metoden returnerer True hvis objektet `account_no` er av ønsket klasse (`int`). Statiske klassemetoder har hverken klassereferanse (`cls`) eller objektreferanse (`self`) som innparameter.

Linje 9 – 11

Her deklarerer de 3 klassevariablene

Linje 13 – 16

En klassemetode som setter ny verdi på `next_account_no`. Legg merke til `cls`, som er en referanse til klassen. Python vil på kallstedet, linje 61, legge inn denne klassereferansen som første argument i kallet på metoden.

Linje 18 – 20

En klassemetode som får tak i antall kontoer som er opprettet

Linje 22 – 24

En klassemetode som får tak i antall transaksjoner som er opprettet

Linje 28 – 30

`__init__()` metoden er endret slik at den henter kontonummer fra klassevariabelen (28), sørger for å telle den opp (29, og oppdaterer variabelen som holder rede på antall kontoer opprettet (30).

Linje 40, 47, 54

Klassevariabel for antall transaksjoner oppdateres.

Linje 71 – 72

Bruker klassemetoder for å hente antall kontoer opprettet og antall transaksjoner opprettet.

Linje 76

Test av den statiske class method `is_valid_account_no`. I kall til statiske klassemetoder sendes det ikke med en klassereferanse (`cls`).

8.10.1 Klassemetoder kan aksesseres via objektreferanse

Det går fint å skrive `acc1.get_account_count()` i stedet for `Account.get_account_count()`.

Det går tydeligere fram at vi kaller en klassemetode når vi skriver `Account.get_account_count()`, så bruk den varianten.

8.11 Operator overloading

Python gir oss mulighet til å definere hva det vil si å utføre aritmetiske, relasjonelle og andre operasjoner på egendefinerte klasser.

Hva skjer egentlig når du slår sammen to strenger med `+`-tegnet i Python, eller adderer to heltall?

```
hello_str = "Hello "
world_str = "World"
total_str = hello_str + world_str
print(total_str) # Hello World

a = 100
b = 200
result = a + b
print(result) # 300
```

Dette fungerer også:

```
total_str = hello_str.__add__(world_str)
result = a.__add__(b)
print(total_str) # Hello World
print(result) # 300
```

Vi får samme utskrift, hva skjer?

Når vi bruker `+`, bytter Python det ut med et kall til metoden `__add__()`. Den venstre operanden blir objektet metoden kalles på, og den høyre operanden sendes inn som argument.

```
print(type(hello_str)) # <class 'str'>
print(type(a))          # <class 'int'>
```

Begge er objekter – `hello_str` er en instans av klassen `str`, og `a` er en instans av `int`.

Hvilke metoder finnes i `int`?

```
print(dir(a))
```

Utskrift (forkortet):

```
[ '__add__', '__sub__', '__mul__', '__truediv__',
  '__eq__', '__lt__', '__gt__', ..., '__str__']
```

Vi ser at `int` har metoder for de fleste vanlige operatorer: `+`, `-`, `*`, `/`, `==`, `<`, `>` osv. Disse kalles **dunder-metoder** (double underscore), og Python bruker dem til å implementere operatorene.

Hvorfor heter det overloading?

I Python betyr *operator overloading* at vi definerer hvordan en operator skal oppføre seg for objekter av en egendefinert klasse. Både `int` og `str` arver fra den innebygde klassen `object`.

```
e = object()
print(dir(e))
print(e)
```

Utskrift (avkortet):

```
[ '__class__', '__eq__', '__str__', ...]
<object object at 0x...>
```

Metoden `__str__()` gir en standard representasjon. Men både `int` og `str` har overlaadet denne metoden, slik at objektene skriver ut en mer lesbar verdi.

8.11.1 Operator overloading i egne klasser

Her er et eksempel med en `Circle`-klasse:

```
01: # file: sc_08_circle_with_op_overload1.py
02: class Circle:
03:     def __init__(self, radius=1):
04:         self._radius = radius
05:
```

```

06:     def __str__(self) -> str:
07:         return f'Radius = {self._radius}'
08:
09:     def __eq__(self, other) -> bool:
10:         return self._radius == other._radius
11:
12:     def __lt__(self, other) -> bool:
13:         return self._radius < other._radius
14:
15:     def __gt__(self, other) -> bool:
16:         return self._radius > other._radius
17:
18:
19: def main():
20:     circle1 = Circle()
21:     circle2 = Circle(25)
22:
23:     print(circle1) # __str__()
24:     print(circle2)
25:
26:     print(f"Are equal? {circle1 == circle2}") # __eq__()
27:     print(f"Is c1 > c2? {circle1 > circle2}") # __gt__()
28:     print(f"Is c1 < c2? {circle1 < circle2}") # __lt__()
29:
30:
31: if __name__ == "__main__":
32:     main()

```

Klassen `Circle` overloader `__eq__()`, `__lt__()` og `__gt__()` metodene, sånn at vi kan sammenligne `Circle` objekter på samme måte som vi sammenligner `int` og `str` objekter.

Utskrift:

```

Radius = 1
Radius = 25
Are equal? False
c1 > c2? False
c1 < c2? True

```

8.11.1.1 Overloade indeksering og slicing: `__getitem__()`

Operatoren `[]` brukes til:
 Å opprette lister

1. Indeksing i lister og strenger
2. Oppslag i dictionaries
3. Slicing

For å overloade denne operatoren bruker vi `__getitem__()`:

```
# file: sc_08_overload_getitem1.py
class Ordre:
    def __init__(self, handlekurv, kunde):
        self._handlekurv = list(handlekurv)
        self._kunde = kunde

    def __getitem__(self, key):
        return self._handlekurv[key]

ordre = Ordre(['Zalo', 'Eple', 'Deodorant'], 'Hansen')

print(ordre[0])          # Zalo
print(ordre[-1])         # Deodorant
print(ordre[0:2])         # ['Zalo', 'Eple']
```

Python genererer et slice-objekt når slicing brukes, og sender det som key til `__getitem__()`.

`self._handlekurv` er en liste, og vi overlater til list klassen å tolke hva key er.

List klassen har overladet `__getitem__()`, og sjekker på om key er en vanlig indeks, eller om det er et slice objekt som må tolkes mer omstendelig.

Så kallet `self._handlekurv[key]` vil i vårt tilfelle se slik ut:

```
self._handlekurv.__getitem__(slice(0, 2, None))
```

8.11.1.2 in-operatoren: `__contains__()`

Vi kan overloade in ved å implementere `__contains__()`:

```
class Ordre:
    def __init__(self, handlekurv, kunde):
        self.handlekurv = list(handlekurv)
        self.kunde = kunde
```

```

def __getitem__(self, key):
    return self.handlekurv[key]

def __contains__(self, key):
    return key in self.handlekurv

ordre = Ordre(['Zalo', 'Eple', 'Deodorant'], 'Hansen')

if 'Zalo' in ordre:
    print("Zalo er i handlekurven") # Zalo er i
handlekurven

```

Merk: Hvis `__contains__()` ikke er definert, vil Python forsøke å bruke `__getitem__()` eller `__iter__()` for å avgjøre om elementet finnes.

Oppsummering: vanlige Operator-metoder

Operator	Metode
+	<code>__add__</code>
-	<code>__sub__</code>
*	<code>__mul__</code>
/	<code>__truediv__</code>
==	<code>__eq__</code>
<	<code>__lt__</code>
>	<code>__gt__</code>
[]	<code>__getitem__</code>
in	<code>__contains__</code>
<code>str()</code>	<code>__str__</code>

© 2025 [Frode Næsje]

Alle rettigheter reservert. Ingen deler av denne publikasjonen kan reproduseres, distribueres, eller overføres i noen form eller på noen måte, elektronisk, mekanisk, fotokopiering, opptak, eller på annen måte, uten forhåndstillatelse fra forfatteren.