Kapittel 13: Mer om funksjoner

Closures, decorators, descriptors, property

Funksjoner er *first class objects*

- Hva betyr det?
  - Python er et fullt ut objektorientert språk. Alt av data som vi oppretter er objekter, basert på en klasse. Alle objekter som vi oppretter er såkalte «first class objects», hvilket betyr at de kan:
  - Lagres i datastrukturer som lister, tupler eller dictionaries
  - Sendes som argumenter til funksjoner
  - Returneres fra funksjoner
  - Tildeles til variabler

Funksjoner som parameter til funksjon

Mange av de tilgjengelige innebygde funksjonene / metodene i Python tar *funksjoner* som parametre, blant annet `sort()` og `sorted()`, hvor du kan sende med en funksjon som bestemmer sorteringsrekkefølgen på det som skal sorteres:

```
data = [[1, 'banana'], [2, 'apple'], [3, 'cherry']]

# Lag en sorteringsfunksjon som sorterer på andre element
def sort_by_second_element(item):
    return item[1]

# Bruk funksjonen sammen med sorted()
sorted_data = sorted(data, key=sort_by_second_element)
# sorted_data = sorted(data, key=lambda x : x[1])

print(sorted_data)
```

Merk: *adressen* til funksjonen

..eller bruk anonym funksjon: `lambda`

Utskrift:

```
[[2, 'apple'], [1, 'banana'], [3, 'cherry']]
```

Closures

- En funksjon kan skrives *inne i en annen funksjon*. Dette gir opphavet til begrepet *closure*, som betyr at den indre funksjonen har tilgang til det som er i dets scope, selv etter at den ytre funksjonen har fullført kjøringen.
- Closures brukes aktivt i mange Python-mekanismer
  - *Decorators* bygger på closures for å "wrappe" funksjoner.
  - Callback-funksjoner i GUI-biblioteker som Tkinter bruker closures for å huske kontekst - for eksempel hvilken knapp eller bruker en event handler tilhører.

Eksempel 1, indre funksjon husker ytre miljø

```
def outer(x: int):
    outer_var = x * 2

    def inner():
        # inner har tilgang til outer sine variabler
        # selv etter at outer har returnert
        inner_var = outer_var * 2
        print(f"x: {x}")
        print(f"outer_var: {outer_var}")
        print(f"inner_var: {inner_var}")
        print()

    return inner  # Returnerer inner adresse
```

```
# Klientkode
print("First closure (x=42):")
closure1 = outer(42)
closure1()  # Printer: 42, 84, 168

print("Second closure (x=10):")
closure2 = outer(10)
closure2()  # Printer: 10, 20, 40

print("Calling first closure again:")
closure1()  # Fortsatt: 42, 84, 168
```

Eksempel 2, closure kan ha flere indre funksjoner (1)

- Det neste eksempelet viser en funksjon `make_account()` med en saldo, og som har tre indre funksjoner som jobber på denne saldoen (`balance`).
  - Funksjonsobjektet gjør her samme jobb som en klasse.
  - Faktisk innkapsler funksjonsobjektet variabelen `balance` bedre enn en klasse - variabelen er ikke tilgjengelig annet enn for `make_account()` og de indre funksjonene.

- Python nøkkelordet `nonlocal` brukes i slike tilfeller som her: en variabel som deklareres i ytre scope og skal *modifiseres* (read/write) fra det indre scopet.

- Variabelen `balance` er ikke global (så derfor kan vi ikke bruke `global`), men `balance` *tilhører den ytre funksjonen*. Den ligger der for at den skal beholde sin tilstand mellom funksjonskall.

Eksempel 2, closure kan ha flere indre funksjoner (2)

```
def make_account(initial_balance: float = 0):
    """Lager et bankkonto-objekt med closure."""
    balance = initial_balance  # Privat variabel
    def deposit(amount: float):
        nonlocal balance
        balance += amount
        return balance
    def withdraw(amount: float):
        nonlocal balance
        if amount > balance:
            print("Insufficient funds!")
            return balance
        balance -= amount
        return balance
    def get_balance():
        return balance
    # Returner funksjonene direkte som tuple
    return deposit, withdraw, get_balance
```

```
# Klientkode
# Bruk account - pakk ut funksjonene
deposit, withdraw, get_balance = make_account(1000)
print("\n--- Bank Account Example ---")
print(f"Initial balance: {get_balance()}")
print(f"After deposit 500: {deposit(500)}")
print(f"After withdraw 200: {withdraw(200)}")
print(f"Final balance: {get_balance()}")

# Prøv å ta ut for mye
withdraw(2000)
```

Utskrift:

```
--- Bank Account Example ---
Initial balance: 1000
After deposit 500: 1500
After withdraw 200: 1300
Final balance: 1300
Insufficient funds!
```

Eksempel 3, closure som *wrapper* (1)

- En *wrapper* lar oss legge til ekstra oppførsel (før, etter, eller rundt) en funksjon, mens originalfunksjonen forblir uendret.
- Eksempelet viser hvordan en closure kan brukes til å "wrappe" (pakke inn) en funksjon med ekstra funksjonalitet.

Eksempel 3, closure som *wrapper* (2)

```
def simple_decorator(func):
    def wrapper():
        print("--- Før funksjonen kalles ---")
        func()  # Closure: wrapper husker func
        print("--- Etter funksjonen er kalt ---")
    return wrapper


def greet():
    print("Hei!")


# Manuell "decorering" - erstatter greet med wrapper-versjonen
print("Original greet:")
greet()


print("\nNå 'decorerer' vi greet:")
greet = simple_decorator(greet)  # greet er nå wrapper-funksjonen
greet()  # Kaller wrapper, som kaller den originale greet
```

`simple_decorator()` tar en funksjon (`func`) som parameter.

Inne i `simple_decorator()` defineres en ny funksjon `wrapper()` som:

1. Skriver ut en melding før `func()` kalles
2. Kaller den originale funksjonen `func()`
3. Skriver ut en melding etter `func()` er kalt

Utskrift:

```
Original greet:
Hei!

Nå 'decorerer' vi greet:
--- Før funksjonen kalles ---
Hei!
--- Etter funksjonen er kalt ---
```

Decorators og @ annotasjonen

- En *decorator* er en funksjon som tar inn en annen funksjon og utvider eller endrer dens oppførsel uten å endre selve funksjonen.
- Decorators er et kraftig verktøy for å legge til funksjonalitet på en gjenbrukbar og lesbar måte.
- En decorator er egentlig det vi har sett på nettopp; - en *closure* - den ytre funksjonen (decorator) returnerer en indre funksjon (wrapper) som "husker" den originale funksjonen fra det ytre scopet.
- Dette gjør at wrapper kan kalle og manipulere den originale funksjonen selv etter at decorator-funksjonen er ferdig kjørt.
- `@`-syntaksen er bare en kortere og mer lesbar måte å skrive kode som vi så på i forrige lysbilde
- Vi skal se på hvordan mekanismen fungerer og hvordan vi kan lage våre egne decorators.

Siste eksempel, erstattet med @

```
def simple_decorator(func):
    def wrapper():
        print("--- Før funksjonen kalles ---")
        func()  # Closure: wrapper husker func
        print("--- Etter funksjonen er kalt ---")
    return wrapper


@simple_decorator
def greet():
    print("Hei!")

greet()

print("\n" + "="*30 + "\n")

@simple_decorator
def say_goodbye():
    print("Ha det!")

say_goodbye()

print("\n" + "="*30 + "\n")
```

`@simple_decorator()` over `greet()`-definisjonen betyr at Python automatisk gjør

```
greet =simple_decorator(greet)
```

etter at funksjonen er definert. Dette er closure-mekanismen fra sc_13_03b_closure_as_wrapper.py), bare med en kortere syntaks.

Utskrift:

```
--- Før funksjonen kalles ---
Hei!
--- Etter funksjonen er kalt ---

========================

--- Før funksjonen kalles ---
Ha det!
--- Etter funksjonen er kalt ---

========================
```

Obs

```
# decorator funksjonen må være definert før bruk av @
def my_cool_wrapper(func):
    def inner():
        print("Before")
        func()
        print("After")
    return inner

@my_cool_wrapper  # må matche decorator func navnet
def hello():
    print("Hello!")

# Det som skjer er at @my_cool_wrapper over funksjonsdefinisjonen betyr:
hello = my_cool_wrapper(hello)
```

Oppsummering decorators

- Decorators (closure-baserte) følger dette mønsteret:
  - En ytre funksjon (decorator) tar en funksjon som parameter
  - Inne i decorator defineres en wrapper-funksjon som legger til ekstra funksjonalitet
  - Wrapper kaller den originale funksjonen
  - Decorator returnerer wrapper
  - @decorator_navn over en funksjon er en kort måte å si "erstatt funksjonen med den wrapped'e versjonen"

@property, og property klassen

- Vi har tidligere brukt `@property` for å lage attributter til en klasse. `@property` er ***ikke en closure basert decorator***, den «wrapper» ikke en funksjon.
- `@property` er en ***klasse-basert* decorator**. Den lager et *descriptor-objekt med* @-syntaksen.

- Vi kan se på en descriptor som et design pattern for å kontrollere attributt-aksess.
- `property` klassen (ja med *liten* p…) implementerer dette design pattern.

property under panseret

Denne protokollen består av tre spesielle metoder:

- __get__(self, instance, owner)
  - Kalles når noen ***leser*** en attributt (f.eks. person.age)
  - Returnerer verdien som skal returneres

- __set__(self, instance, value)
  - Kalles når noen **skriver** til en attributt (f.eks. person.age = 30)
  - Returnerer typisk ingenting, men kan kaste exception ved ugyldig verdi

- __delete__(self, instance)
  - Kalles når noen ***sletter*** en attributt (f.eks. del person.age)
  - Returnerer typisk ingenting

Kodeeksempel: opprette og bruke et property objekt

```
# Circle klasse med property, men ikke @property decorator
class Circle:
    def __init__(self, radius):
        self.radius = radius # setter called, creates _radius attribute

    def _set_radius(self, radius): # setter for radius
        if radius < 0 or radius > 99:
            raise ValueError(f'Invalid radius {radius}')
        self._radius = radius # create _radius attribute

    def _get_radius(self): # Getter for radius
        return self._radius

    radius = property(_get_radius, _set_radius) # Lag property radius
```

```
# Klientkode
try:
    c1 = Circle(10)
    print(c1.radius) # _get_radius() blir kalt
    c1.radius = 20 # _set_radius() blir kalt
    c1.radius = -2 # _set_radius() blir kalt
except ValueError as ex:
    print("Exception: ",ex)
```

Vi har laget en ***property*** `radius` ved hjelp av denne kodelinja.
Denne oppfører seg som en attributt.

Når vi *setter* denne attributten vil metoden `_set_radius()` bli kalt.

Når vi kun skal hente verdi blir `_get_radius()` kalt, *automagisk*.

Utskrift:

```
10
Exception:  Invalid radius -2
```

Kodeeksempel: bruke @property decorator

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def radius(self): # Getter for radius
        return self._radius

    @radius.setter # Setter for radius
    def radius(self, radius):
        if radius < 0 or radius > 99:
            raise ValueError(f"Invalid radius {radius}")
        self._radius = radius
```

```
# Klientkode
try:
    c1 = Circle(10)
    print(c1.radius) # property-getter blir kalt
    c1.radius = 20  # property-setter blir kalt
    c1.radius = -2  # property-setter blir kalt
except ValueError as ex:
    print("Exception: ",ex)
```

Her bruker vi `@property` og `@radius.setter`. Bak kulissene vil notasjonen sørge for å opprette et `property` objekt med implementasjonene av `_set_radius()` og `_get_radius()` som vi så i forrige kodeeksempel.

Utskrift:

```
10
Exception:  Invalid radius -2
```

Kodeeksempel: lage read-only attributter

```
class Circle:
    def __init__(self, radius):
        # Validering ved konstruksjon (read-only etterpå)
        if radius < 0 or radius > 99:
            raise ValueError(f"Invalid radius {radius}")
        self._radius = radius

    @property
    def radius(self):  # Getter for radius (read-only)
        return self._radius
```

```
# Klientkode
try:
    c1 = Circle(10)
    print(c1.radius)  # property-getter blir kalt
    # Forsøk på å sette ny radius feiler fordi
property er read-only
    c1.radius = 20
except AttributeError as ex:
    print("AttributeError:", ex)
# Ugyldig konstruksjon (validering i __init__)
try:
    c2 = Circle(-2)
except ValueError as ex:
    print("ValueError:", ex)
```

Vi kan enkelt lage read only attributter ved å sløyfe setteren. I vårt tilfelle må vi passe på å flytte valideringen til `__init__()`

Utskrift:

```
10
AttributeError: can't set attribute 'radius'
ValueError: Invalid radius -2
```