13 Mer om funksjoner: closures, decorators, descriptors

13.1 Funksjoner er også «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

13.2 Funksjon som parameter til funksjon

Det interessante er at også *funksjoner* er «first class objects», altså de kan blant annet være parametre til funksjoner, og returneres fra funksjoner. 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:

```
# Lag en liste i liste
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)

print(sorted_data)
```

Utskrift:

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

Legg merke til at i linja

```
sorted_data = sorted(data, key=sort_by_second_element)
```

..så er `sort_by_second_element` altså navnet på funksjonen uten paranteser. Effekten er som vi har sett i tidligere eksempler at *adressen* til funksjonen sendes til `sorted`, og implementasjonen av `sorted` kaller denne funksjonen når den sorterer, altså det sorteres på det andre elementet i sublista.. Dette demonstrerer at en funksjon er et *first class object*; - funksjonen sendes som *parameter* til en annen funksjon.

13.3 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 (neste delkapittel) 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.

13.3.1 Eksempel 1, indre funksjon husker ytre miljø

I eksempelet nedenfor er `inner()` en closure, fordi den har tilgang til variablene `outer_var` og parameteren `x` til `outer()`, *selv etter at `outer()` har fullført kjøringen*:

```
# file: sc_13_02_inner_outer.py
# Closures - indre funksjoner som "husker" det ytre miljøet

# Eksempel 1: Grunnleggende closure
def outer(x: int):
    outer_var = x * 2

    def inner():
```

```
        # Closure: inner har tilgang til outer sine variabler
        # selv etter at outer har returnert
        inner_var = outer_var * 2
        print(f"x: {x}")           # Fra outer sin parameter
        print(f"outer_var: {outer_var}")  # Fra outer sin lokale
variabel
        print(f"inner_var: {inner_var}")  # inner sin egen
variabel
        print()

    return inner  # Returnerer inner adresse

# Hver closure "husker" sitt eget miljø
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

# Begge closures eksisterer samtidig med sine egne verdier
print("Calling first closure again:")
closure1()  # Fortsatt: 42, 84, 168
```

Utskrift:

```
First closure (x=42):
x: 42
outer_var: 84
inner_var: 168

Second closure (x=10):
x: 10
outer_var: 20
inner_var: 40

Calling first closure again:
x: 42
outer_var: 84
inner_var: 168
```

Forklaring:

Legg merke til at `inner()` sin kode er skrevet inne i `outer()`. `inner()` har tilgang til `outer()` sine parametre og lokale variabler.

Dette kalles en ***closure*** - den indre funksjonen "husker" variablene fra det ytre scopet, og disse lever så lenge det eksisterer en referanse til funksjonsobjektet.

Siste linje i `outer()` funksjonen returnerer adressen til `inner()`.

Dette:
```
closure1 = outer(42)
closure1()
```

kaller `outer()` som returnerer adressen til `inner()`, som tilordnes `closure1`. `closure1()` er da et kall til `inner()`, som eksekveres og vil skrive ut:
```
x: 42
outer_var: 84
inner_var: 168
```

Den siste linja med kode:
```
closure1()  # Fortsatt utskrift 42, 84, 168
```

Viser at funksjonsobjektet (inner) eksisterer når kall nummer to til `closure1()` utføres, variablene lever ennå, «fanget» i closuren.

13.3.2Eksempel 2, closures kan ha flere indre funksjoner

Det neste eksempelet viser en funksjon `make_account()` med en saldo, og som har tre indre funksjoner som jobber på denne saldoen (`balance`). Funksjonsobjektet kan man her si gjør 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 under: 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 tilhører den ytre funksjonen. Den ligger der for at den skal beholde sin tilstand mellom funksjonskall.

```
# file: sc_13_03a_account_as_closure.py
# closure med flere funksjoner

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

# 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 en funksjon

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. Dette er grunnlaget for decorators i neste delkapittel.

```
# file: sc_13_03b_closure_as_wrapper.py
# Closure som wrapper en funksjon

def simple_decorator(func):
    # Tar en funksjon,returnerer en ny funksjon som wrapper den.
    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

# Dette er presis det som skjer når vi bruker @-syntaksen i
# kommende delkapitel om decorators
```

Forklaring:

`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

`wrapper()` er som i eksempel 1 også en **closure** - den "husker" `func` fra det ytre scopet (`simple_decorator()`).

`simple_decorator()` returnerer adressen til `wrapper()`.

Først kaller vi den originale `greet()`:

```
greet()  # Skriver bare ut: Hei!
```

Så "decorerer" vi `greet()` manuelt:

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

Dette kallet til `simple_decorator(greet)` returnerer wrapper-funksjonen, som tilordnes variabelen `greet`. Nå er `greet` ikke lenger den originale funksjonen, men wrapper-funksjonen som "husker" den originale `greet`.

Når vi nå kaller:

```
greet()
```

kjøres faktisk `wrapper()`, som skriver ut:

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

`wrapper()` kaller den originale `greet()` (fanget i closuren), men legger til ekstra funksjonalitet før og etter.

Dette mønsteret - å erstatte en funksjon med en wrapped versjon - er nøyaktig det Python gjør *automatisk* når vi bruker @-syntaksen (decorator-syntaksen).

13.4Decorators 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 kapittel om closures - dette kapittelet viser hvordan mekanismen fungerer og hvordan vi kan lage våre egne decorators.

Under ser vi et kodeeksempel som demonstrerer decorator-konseptet gjennom tre steg, fra det mest grunnleggende til mer praktiske anvendelser. Merk at den manuelle decorator-mekanismen (uten `@`) allerede er vist i `sc_13_03b_closure_as_wrapper.py`, så vi går rett på `@`-syntaksen her.

Steg 1

viser `@`-syntaksen i praksis: Hvordan `@decorator_navn` plassert over en funksjonsdefinisjon automatisk "wrapper" funksjonen. En kommentar minner om at `@simple_decorator` er det samme som `greet = simple_decorator(greet)`, som er closure-mekanismen vi allerede har sett.

Steg 2

utvider decorator til å håndtere funksjoner med argumenter og returverdier. Ved å bruke `*args` og `**kwargs` kan én decorator fungere med alle typer funksjoner, uansett hvor mange argumenter de har. Dette steget viser også hvordan man bevarer returverdier gjennom decorator-laget.

Steg 3

demonstrerer en praktisk anvendelse: en decorator som teller hvor mange ganger en funksjon kalles. Dette illustrerer hvordan decorators

kan holde på tilstand (state) mellom funksjonskall ved hjelp av closures og `nonlocal`.

Kode Steg 1:

```
01: # file: sc_13_04_decorator.py
02: # Decorators - funksjoner som wrapper andre funksjoner
03: # (Dette bygger på closure fra
sc_13_03b_closure_as_wrapper.py)
04:
06: def simple_decorator(func):
07:     """Wrapper en funksjon med ekstra funksjonalitet."""
08:     def wrapper():
09:         print("--- Før funksjonen kalles ---")
10:         func()  # Closure: wrapper husker func
11:         print("--- Etter funksjonen er kalt ---")
12:     return wrapper
13:
14: # @simple_decorator betyr: greet = simple_decorator(greet)
15: @simple_decorator
16: def greet():
17:     print("Hei!")
18:
19: greet()
20:
21: print("\n" + "="*50 + "\n")
22:
23: @simple_decorator
24: def say_goodbye():
25:     print("Ha det!")
26:
27: say_goodbye()
28:
29: print("\n" + "="*50 + "\n")
```

Forklaring til kode Steg 1: Enkel decorator med @-syntaks

Linje 6-12:

Vi definerer en `simple_decorator()` som tar en funksjon `func` som parameter. Inni lager vi en ny funksjon `wrapper()` som:

- Skriver ut en melding før funksjonen kalles

- Kaller den originale funksjonen `func()` (closure: wrapper husker `func`)

- Skriver ut en melding etter funksjonen er kalt

`simple_decorator()` returnerer `wrapper`-funksjonen, ikke resultatet av et funksjonskall.

Linje 15-18:

`@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.

Linje 20:

Når vi kaller `greet()`, kjører vi faktisk `wrapper()`, som igjen kaller den originale `greet`-funksjonen, men med ekstra funksjonalitet før og etter.

Linje 24-26:

Et andre eksempel viser at samme decorator kan brukes på flere funksjoner.

Kode Steg 2:

```
30:
31: # Steg 2: Decorator som håndterer argumenter og returverdier
32: def smart_decorator(func):
33:     """Decorator med argumenter og returverdier."""
34:     def wrapper(*args, **kwargs):
35:         print(f"Kaller {func.__name__} med args={args},
kwargs={kwargs}")
36:         result = func(*args, **kwargs)
37:         print(f"{func.__name__} returnerte: {result}")
38:         return result
39:     return wrapper
40:
41: @smart_decorator
42: def add(a, b):
43:     return a + b
44:
45: @smart_decorator
46: def greet_person(name, greeting="Hei"):
47:     return f"{greeting}, {name}!"
48:
49: result1 = add(5, 3)
```

```
50: print(f"Resultat: {result1}")
51:
52: print()
53:
54: result2 = greet_person("Anna", greeting="Hallo")
55: print(f"Resultat: {result2}")
56:
57: print("\n" + "="*50 + "\n")
```

Forklaring til kode Steg 2: Decorator med argumenter og returverdier

Linje 31-39:

`smart_decorator` er mer avansert.

Nøkkelpunkter:

-Linje 33: `wrapper(*args, **kwargs)` - bruker `*args` og `**kwargs` for å ta imot alle mulige argumenter (både posisjonelle og navngitte).

- Linje 34: Vi logger funksjonsnavnet (`func.__name__`) og argumentene som ble sendt inn.

- Linje 35: Vi kaller den originale funksjonen med alle argumentene og lagrer resultatet.

- Linje 36: Vi logger returverdien.

- Linje 37: Viktig! Vi må returnere resultatet, ellers vil funksjonen alltid returnere `None`.

Linje 41-46:

To eksempler på funksjoner med ulike signaturer:

- `add(a, b)` - enkel funksjon med to argumenter

- `greet_person(name, greeting="Hei")` - har både obligatorisk og navngitt argument med default-verdi

Begge fungerer med samme decorator fordi wrapper bruker `*args, **kwargs`.

Linje 48-54:

Vi ser at funksjonene fungerer normalt, men med ekstra logging. Returverdiene bevares og kan brukes videre.

Kode Steg 3:

```
58:
59: # Steg 3: Praktisk eksempel - logging decorator med state
60: def log_calls(func):
61:     """Logger alle kall til funksjonen."""
62:     call_count = 0
63:
64:     def wrapper(*args, **kwargs):
65:         nonlocal call_count
66:         call_count += 1
67:         print(f"[LOG] Kall #{call_count} til
{func.__name__}")
68:         return func(*args, **kwargs)
69:
70:     return wrapper
71:
72: @log_calls
73: def calculate(x, y):
74:     return x * y + x
75:
76: print(calculate(3, 4))
77: print(calculate(5, 2))
78: print(calculate(1, 1))
```

Forklaring til kode Steg 3: Eksempel med state (tilstand)

Linje 59-67:

`log_calls()` viser en decorator med intern tilstand:

- Linje 60:

Vi deklarerer `call_count = 0` i decorator-funksjonen. Denne variabelen "huskes" av wrapper (closure).

- Linje 63: `nonlocal call_count` lar oss modifisere `call_count` fra wrapper.

- Linje 64: Hver gang wrapper kalles, øker vi telleren.

- Linje 65: Vi logger hvilken gang funksjonen kalles og funksjonens navn.

Linje 69-74:

Hver gang vi kaller `calculate()`, ser vi at telleren øker. Dette demonstrerer at decorator kan holde på tilstand mellom funksjonskall.

13.4.1 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 versjonen"

Merk:

```
# 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)
```

13.5 @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.

Denne protokollen består av tre spesielle metoder:

`__get__(self, instance, owner)`

- Kalles når noen ***leser*** en attributt (f.eks. `person.age`)
- `self` er descriptor-objektet selv
- `instance` er objektet vi leser fra (f.eks. `person`), eller `None` hvis vi aksesserer fra klassen
- `owner` er klassen objektet tilhører (f.eks. `Person`)
- Returnerer verdien som skal returneres

`__set__(self, instance, value)`

- Kalles når noen **skriver** til en attributt (f.eks. `person.age = 30`)
- `self` er descriptor-objektet selv
- `instance` er objektet vi skriver til (f.eks. `person`)
- `value` er verdien som skal lagres (f.eks. `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)
- `self` er descriptor-objektet selv
- `instance` er objektet vi sletter fra (f.eks. `person`)
- Returnerer typisk ingenting

13.5.1 Kodeeksempel: opprette og bruke et property objekt

Det følgende kodeeksempelet oppretter en property deskriptor:

```
01: # file: sc_13_05_property_basic.py
02: # Circle klasse med property, men ikke @property decorator
03: class Circle:
04:     def __init__(self, radius):
```

```
05:         self.radius = radius # setter called, creates _radius
attribute
06:
07:     def _set_radius(self, radius): # Setter for radius
08:         if radius < 0 or radius > 99:
09:             raise ValueError(f'Invalid radius {radius}')
10:         self._radius = radius # create _radius attribute
11:
12:     def _get_radius(self): # Getter for radius
13:         return self._radius
14:
15:     radius = property(_get_radius, _set_radius)
16:
17: # Try Circle
18: try:
19:     c1 = Circle(10)
20:     print(c1.radius) # _get_radius() blir kalt
21:     c1.radius = 20 # _set_radius() blir kalt
22:     print(c1.radius) # _get_radius() blir kalt
23:     c1.radius = -2 # _set_radius() blir kalt
24: except ValueError as ex:
25:     print("Exception: ",ex)
```

Utskrift:

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

Forklaring til kode:

Vi har laget en ***property*** `radius` ved hjelp av kodelinja

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

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*.
Merk at for at setteren skal brukes for sjekk av korrekt radius, så må _radius attributten skapes i setteren! Derfor er koden i `__init__()` ***slik som den er:*** `self.radius = radius`

Merk:
Det som ser ut som et funksjonskall `property(...)` er egentlig oppretting av et objekt av klassen `property()`. Denne klassen

implementerer *deskriptor protokollen*, som oversetter dot-operatoren til kall til de setter- og getter metodene som vi har laget.

13.5.2 Kodeeksempel: bruke @property decorator

Følgende kodeeksempel implementerer det samme, men med @property decoratoren:

```
01: # file: sc_13_06_property_with_@property.py
02: # Circle-klasse med @property-decorator
03: class Circle:
04:     def __init__(self, radius):
05:         self.radius = radius
06:
07:     @property
08:     def radius(self): # Getter for radius
09:         return self._radius
10:
11:     @radius.setter # Setter for radius
12:     def radius(self, radius):
13:         if radius < 0 or radius > 99:
14:             raise ValueError(f"Invalid radius {radius}")
15:         self._radius = radius
16:
17: # Try Circle
18: try:
19:     c1 = Circle(10)
20:     print(c1.radius) # property-getter blir kalt
21:     c1.radius = 20  # property-setter blir kalt
22:     c1.radius = -2  # property-setter blir kalt (kaster
ValueError)
23: except ValueError as ex:
24:     print("Exception: ",ex)
```

Utskrift:

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

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.