

6 Funksjoner

Læringsmål dette kapittel:

- Forstå funksjoner og kunne definere en funksjon
- Kunne kalle en funksjon med riktig antall argumenter
- Forstå bruken av `return`-setningen for å sende verdier tilbake fra en funksjon
- Forstå hva som skjer på stack og heap under et funksjonskall
- Forstå overføring av argumenter og forskjellen mellom immutable og mutable datatyper i funksjonskall
- Kunne returnere flere verdier fra en funksjon
- Kjenne til og kunne bruke keyword arguments (navngitte argumenter) for å gjøre funksjonskall mer lesbare og fleksible
- Kunne definere funksjoner med default-verdier (standardverdier) for parametere
- Vite at default-verdier kun evalueres én gang når funksjonen defineres, og forstå implikasjonene av dette, spesielt med mutable objekter som lister og ordbøker (dictionaries)
- Kunne bruke *args-teknikken for å akseptere et vilkårlig antall posisjonsargumenter i en funksjon
- Kunne bruke **kwargs-teknikken for å akseptere et vilkårlig antall navngitte argumenter i en funksjon
- Forstå synlighet (scope) av variabler i Python
- Kjenne til begrepene lokalt, globalt og innebygd namespace.
- Forstå søkerekkefølgen for variabler (LEGB-regelen): Local → Enclosing → Global → Built-in
- Forstå diverse regler om scope i forbindelse med løkker, funksjoner og comprehensions

6.1 Å definere en funksjon

Funksjoner er en av de absolutt viktigste byggeklossene vi har for å kunne lage modulære og lesbare programmer. En funksjon er ikke noe annet enn et navn på en kodesnutt som vi ønsker å bruke gjentatte ganger.

La oss begynne med det helt enkle, sånn at vi ser prinsippet: vi ønsker oss en funksjon som legger sammen to tall.

Først må vi definere funksjonen:

```
def add_two_numbers(a, b):
    return a + b

# bruk den
svar1 = add_two_numbers(1,2)
svar2 = add_two_numbers(3,4)
print(svar1) # 3
print(svar2) # 7
```

Vi lager en funksjon ved å lage en funksjonsheader som må inneholde

- **def** – et nøkkelord som forteller Python at vi definerer en funksjon
- **Funksjonsnavn** – et navn vi velger, som følger navngivingsreglene i Python
- **Parenteser ()** – alltid med, selv om funksjonen ikke tar inn noen verdier
- **Parametre** (valgfritt) – plasseres inni parentesene hvis funksjonen skal ta imot inn-data
- **Kolon :** – avslutter headeren og markerer starten på funksjonsblokka, altså koden til funksjonen

Funksjonsheaderen vår er

```
def add_two_numbers(a, b):
```

Funksjonen *kan returnere* noe, vi bruker `return` setninga for å gjøre dette, i vårt eksempel summen av parametrene a og b.

Vi *kaller* funksjonen ved å oppgi dens navn, samt sende med riktig antall **argumenter**, ett for parameteren a, og ett for parameteren b. Vi kaller funksjonen to ganger ovenfor, og tar vare på resultatet i variablene `svar1` og `svar2`:

```
svar1 = add_two_numbers(1,2)
svar2 = add_two_numbers(3,4)
```

La oss lage en mer nyttig funksjon som gitt en *score* gir en karakter tilbake:

```
def calc_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
```

```

        return "B"
    elif score >= 60:
        return "C"
    elif score >= 50:
        return "D"
    elif score >= 40:
        return "E"
    else:
        return "F"

score1 = 85
grade1 = calc_grade(score1)
print(grade1)

score2 = 92
grade2 = calc_grade(score2)
print(grade2)

```

Utskrift:

B
A

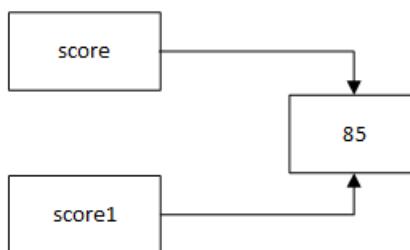
Argumentet, heltallet `score1` (og `score2` i det andre kallet) er som vi har lært en referanse til et objekt av `int` datatypen. I forbindelse med funksjonskallet lages det en *kopi* (parameteren `score`) som refererer samme `int` objekt. Se Figur 6.1.

```

def calc_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 60:
        return "C"
    elif score >= 50:
        return "D"
    elif score >= 40:
        return "E"
    else:
        return "F"

score1 = 85
grade1 = calc_grade(score1)

```



Figur 6.1 Argument og parameter refererer det samme

6.2 Bruk av stack og heap ved funksjonskall

Som programmerer bør du kjenne til hva som skjer ved et funksjonskall.

I et program er flere ulike typer minneområder brukt. Ved et funksjonskall er minneområdene *stack* og *heap* i bruk.

Stack er minneområdet som brukes til å administrere utveksling av data mellom funksjonskallet og funksjonen, mens **heap** brukes til å alllokere plass til objektene som det refereres til.

Når en funksjon kalles, legges følgende på stack (forenklet og i typisk rekkefølge):

1. Returadresse

Hvor programmet skal fortsette å eksekvere etter at funksjonen er ferdig, dette er grovt sett kodelinja etter funksjonskallet

2. Argument-referanser

Referanser til objektene som sendes inn som argumenter

3. Parameter-referanser

Disse peker til de samme objektene som argumentene (i Python kopieres referansen, ikke objektet)

4. Lokale variabler

Alt som defineres inne i funksjonen (inkludert parameterne) lagres i stack-frame for funksjonen

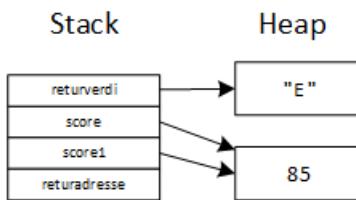
5. Returverdi-referanse

Når funksjonen returnerer, opprettes en referanse til returverdien (som ligger på heap)

Se Figur 6.2

```
def calc_grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 60:
        return "C"
    elif score >= 50:
        return "D"
    elif score >= 40:
        return "E"
    else:
        return "F"

score1 = 85
grade1 = calc_grade(score1)
```



Figur 6.2 Bruk av stack ved et funksjonskall

6.3 Overføring av argumenter: immutable og mutable

6.3.1 Immutable

Vi har tidligere sett at `int` og `str` er eksempler på immutable datatyper – det vil si at de ikke kan endres etter at de er opprettet. Når vi forsøker å "endre" slike verdier i en funksjon, skapes det i virkeligheten et nytt objekt, og den opprinnelige verdien forblir uendret.

For å tydelig vise dette, lager vi to funksjoner: `modify_int(x)` og `modify_str(s)`. Disse forsøker å endre henholdsvis et heltall og en streng som sendes inn som argument.

```
def modify_int(x):
    print("Inside function (before change):", x, "| id:", id(x))
    x = x + 1
    print("Inside function (after change):", x, "| id:", id(x))

def modify_str(s):
    print("Inside function (before change):", s, "| id:", id(s))
    s = s + "!"
    print("Inside function (after change):", s, "| id:", id(s))
```

Vi kaller funksjonene slik:

```
x = 10
modify_int(x)
print("Outside function:", x, "| id:", id(x))

s = "Hello"
modify_str(s)
print("Outside function:", s, "| id:", id(s))
```

Utskriften blir:

```
Inside function (before change): 10 | id: 1754625606160
Inside function (after change): 11 | id: 1754625606192
Outside function: 10 | id: 1754625606160
Inside function (before change): Hello | id: 1754627236528
Inside function (after change): Hello! | id: 1754627240304
Outside function: Hello | id: 1754627236528
```

Observasjon: Objekt-ID-en endres inne i funksjonen, noe som viser at det er opprettet et *nytt* objekt. Den opprinnelige verdien utenfor funksjonen er uendret.

6.3.2 Mutable

`list` er en mutabel datatype. Et tilsvarende eksempel som for `int` og `str` viser forskjellen i oppførsel.

```
def modify_list(lst):
    print("Inside function (before change):", lst, "| id:", id(lst))
    lst.append(4)
    print("Inside function (after change):", lst, "| id:", id(lst))
```

Vi kaller `modify_list` og ser hva som skjer med lista:

```
lst = [1, 2, 3]
modify_list(lst)
print("Outside function:", lst, "| id:", id(lst))
```

Utskrift:

```
Inside function (before change): [1, 2, 3] | id: 1754626995968
Inside function (after change): [1, 2, 3, 4] | id: 1754626995968
Outside function: [1, 2, 3, 4] | id: 1754626995968
```

Observasjon: Objekt-ID-en er den samme før og etter endringen, både inne i og utenfor funksjonen. Det betyr at listen faktisk ble *modifisert direkte*.

6.4 Navngiving argument og parametre

En utbredt nybegynner-misforståelse er at navn på argument og parameter har betydning for parameteroverføring. Det er absolutt ikke tilfelle. Parameter navn er kun kjent innenfor funksjonen og konflikter ikke med variabelnavn definert i globalt scope. Derfor kan parameter navn være lik argument navn, det er ingen konflikt.

Denne kodesnutten demonstrerer dette:

```
# file: sc_06_03.py
01: # parameter navn kan være hva som helst
02: # parameter navn har lokalt scope og har ingenting
03: # med variabler utenfor funksjonen å gjøre
```

```

04: def add_two_numbers(x, y):
05:     return x + y
06:
07: def add_two_numbers(a, b):
08:     return a + b
09:
10:
11: x = 1
12: y = 2
13: svar1 = add_two_numbers(x, y)
14: print(svar1) # 3
15: svar1 = add_two_numbers(x, y)
16: print(svar1) # 3

```

6.5 Retur av flere verdier

En funksjon kan returnere mer enn én verdi. Dette er nyttig når vi ønsker å hente ut flere resultater fra én beregning. For eksempel hvis vi ønsker å finne både summen og gjennomsnittet av tre tall:

```

def calculate_sum_and_average(a, b, c):
    total = a + b + c
    average = total / 3
    return total, average

sum1, avg1 = calculate_sum_and_average(3, 6, 9)
print("Sum:", sum1)
print("Gjennomsnitt:", avg1)

```

Funksjonen returnerer to verdier, og de blir «pakket ut» i to variabler. Dette gjør koden ryddig og effektiv.

6.6 Type hints

Type hints i Python er en måte å spesifisere hvilke datatyper funksjoner *forventer* som argumenter og hva de returnerer. Dette gjør koden mer lesbar og lettere å forstå, og kan hjelpe med feilsøking og automatisk verktøystøtte (f.eks. i IDE-er).

Her er funksjonen `add_two_numbers` med type hints lagt til:

```
def add_two_numbers(a: int, b: int) -> int:
    return a + b
```

Forklaring:

- `a: int` og `b: int` betyr at funksjonen forventer to heltall som argumenter.
- `-> int` betyr at funksjonen returnerer et heltall.

Du kan også bruke andre typer, som float, str, list, dict.

Type hints vil *ikke hindre at du kaller funksjonen med feil type argumenter*, det er kun et «tips» om hva funksjonen *forventer* og hva den returnerer.

6.7 Keyword arguments

Keyword arguments gjør det mulig å sende inn verdier til funksjonen ved å bruke *navn* på parameterne. Dette gjør koden mer lesbar, spesielt når funksjonen har mange parametere.

```
def create_greeting(name, greeting):
    return f"{greeting}, {name}!"

msg1 = create_greeting(name="Ola", greeting="Hei")
msg2 = create_greeting(greeting="Hallo", name="Kari")
print(msg1)
print(msg2)
```

Ved å bruke navn på argumentene på kallstedet, kan vi sende dem *i hvilken som helst rekkefølge*. Dersom vi ikke bruker keyword, så må vi sende argumentene i nøyaktig den rekkefølgen de er forventet å komme.

6.8 Default verdier

Noen ganger ønsker vi at en funksjon skal ha en *standardverdi* dersom vi ikke sender inn noe. Dette gjør funksjonen mer fleksibel.

```
def greet(name, greeting="Hei"):
    print(f"{greeting}, {name}!")

greet("Anna")           # Bruker standardverdi
greet("Jon", "Hallo")   # Overstyrer standardverdi
```

Standardverdier gjør at vi kan kalle funksjonen med færre argumenter når det passer.

Dersom du både har standardverdi og ikke for parametrerne, så må parametrerne uten standardverdi komme før de med standardverdi:

Ok:

```
def funksjon(a, b=2, c=3):
    return a + b + c
```

Ikke ok:

```
def funksjon(a=1, b): # du får SyntaxError
    return a + b
```

OBS! standardverdier evalueres kun en gang, når funksjonen defineres!

Standardverdier evalueres kun når funksjonen *defineres*, ikke hver gang funksjonen kalles. Dette kan føre til uventet oppførsel med *mutable* objekter som *list* og *dict*:

```
def legg_til(element, liste=[]):
    liste.append(element)
    return liste

print(legg_til(1)) # [1]
print(legg_til(2)) # [1, 2] !
```

Det er den samme lista som brukes ved kall nummer to!

Grunnen til dette er at Python lagrer standardverdiene som en del av funksjonsobjektet (vi kommer tilbake til funksjonsobjekter senere).

Det er en ytelsesoptimalisering – i stedet for å evaluere standardverdien hver gang funksjonen kalles, evalueres den én gang og gjenbrukes.

Mutable objekter som **lister**, **dictionaries** og **set** kan endres etter at de er opprettet. Hvis du bruker en slik som standardverdi, og endrer

den inne i funksjonen, vil endringen *vedvare* til neste gang funksjonen kalles.

Immutable objekter som **int**, **float**, **str**, **tuple** kan *ikke endres* etter at de er opprettet. Hvis du bruker en slik som standardverdi, er det ingen risiko for at verdien "husker" tidligere kall, fordi den ikke kan endres.

Løsning hvis mutable datatyper:

```
def legg_til(element, liste=None):
    if liste is None:
        liste = []
    liste.append(element)
    return liste

print(legg_til(1))  # [1]
print(legg_til(2)) # [2]
```

6.9 *args teknikken – flere posisjonsargumenter

Med ***args (arguments)** kan vi sende inn et vilkårlig antall argumenter til en funksjon. Dette er nyttig når vi ikke vet på forhånd hvor mange tall vi skal summere.

```
def total_sum(*numbers):
    total = 0
    for n in numbers:
        total += n
    return total

print(total_sum(1, 2, 3))
print(total_sum(10, 20, 30, 40))
```

***args** samler alle argumentene i en **tuple**, som vi kan iterere over.

6.10 **kwargs teknikken – flere keyword argumenter

(*Dette blir mer meningsfull info når vi har lært om dictionaries, om litt ☺)*

Med `**kwargs` (**keyword arguments**) kan vi sende inn et vilkårlig antall navngitte (keyword) argumenter. Dette er nyttig når vi vil vise informasjon som kan variere.

```
def print_user_info(**info):
    for key in info:
        print(key, ":", info[key])

print_user_info(name="Lise", age=30)
print_user_info(name="Per", hobby="ski", city="Oslo")
```

Utskrift:

```
name : Lise
age : 30
name : Per
hobby : ski
city : Oslo
```

`**kwargs` samler alle navngitte argumenter i en dictionary. Vi kan da bruke en løkke for å vise dem.

6.11 Synlighet (scope) av variabler

Når vi programmerer i Python, er det viktig å forstå *scope* – altså hvor i programmet en variabel er synlig og kan brukes. I dette delkapitelet tar vi for oss scope på generell basis, altså ikke bare i funksjoner.

6.11.1 Scope og namespace

Scope beskriver *hvor i koden en variabel er gyldig og tilgjengelig*. Det handler om synlighet – altså hvilke deler av programmet som kan bruke et gitt navn.

Typer scope i Python:

- **Lokal:** Gyldig bare inne i en funksjon, klasse eller annen struktur som skaper lokalt scope.
- **Global:** Gyldig i hele modulen eller skriptet, fra det punktet den er definert.

- **Ikke-lokal (enclosing):** Gyldig i en omsluttende funksjon (ved næstede funksjoner).
- **Innebygd (built-in):** Gyldig overalt – navn som print, len, osv.

Mens **scope** forteller hvor en variabel eller funksjon er *tilgjengelig*, så forteller **namespace** hva variablene / funksjonen refererer til, altså hvor objektene fysisk befinner seg.

Python har flg typer namespace:

1. *Lokalt namespace:* Inne i funksjonen, klassen eller modulen der variablen er definert
2. *Enclosing namespace:* I det omsluttende (nested) funksjonsomfanget, f.eks. i en ytre funksjon hvis du har en næstet funksjon.
3. *Globalt namespace:* Navn som er definert på øverste nivå i programmet, tilgjengelig i hele modulen eller skriptet
4. *Built-in (innebygget) namespace:* Navn som alltid er tilgjengelige, som `print()` og `len()`.

Når Python ser etter en variabel, søker den i denne rekkefølgen: Local → Enclosing → Global → Built-in.

Følgende tabell viser hva som skaper lokalt scope i Python, *kanskje det viktigste å legge merke til er at if, for, while og try ikke skaper lokalt scope:*

Hva	Skaper lokalt scope?	Kommentar
<code>def</code> (funksjon)	Ja	Lokalt scope for parametere og lokale variabler
<code>class</code>	Ja	Eget scope for attributter og metoder, <i>men disse er likevel tilgjengelig for klientkode</i>
Modul (.py-fil)	Ja	Globalt scope for modulen
<code>lambda</code>	Ja	Som funksjon – har eget scope

Hva	Skaper lokalt scope?	Kommentar
list comprehension	Ja	Variabler inni er ikke tilgjengelige utenfor
dict/set comprehension	Ja	Samme som list comprehension
Generator expression	Ja	Eget scope
if, for, while, try	Nei	Variabler lever videre utenfor blokken
with-blokk	Nei	Ingen nytt scope
match-blokk (Python 3.10+)	Nei	Ingen nytt scope

6.11.2 Detaljer om scope i Python

Løkker og betingelser skaper ikke eget scope: I motsetning til mange andre språk, lager ikke for, while, if, eller try-blokker i Python et nytt scope. Variabler definert i disse blokkene tilhører det samme omfanget som koden rundt dem.

```
if True:
    y = 10
print(y) # Utskrift: 10, y er global
```

6.11.3 Funksjoner skaper lokalt scope

Variabler definert *inne i en funksjon* er *lokale* med mindre de deklarereres som *global* eller *nonlocal* (for næstede funksjoner).

```
def func():
    z = 42 # Lokal variabel
```

```
func()
print(z) # Feil: NameError, z er ikke definert
```

I Python kan vi ha nøstede funksjoner, det vil si funksjoner i funksjoner. Hvis du er inne i en nøstet funksjon kan du bruke `nonlocal`-nøkkelordet for å referere til en variabel i det *omsluttende* omfanget:

```
def outer():
    count = 0
    def inner():
        nonlocal count
        count += 1
    inner()
    print(count) # Utskrift: 1
outer()
```

6.11.4 Comprehensions skaper lokalt scope

Fra Python 3 skaper *comprehensions* sitt eget scope, så variabler definert i en comprehension lekker ikke ut til det ytre omfanget:

```
a_list = [i for i in range(5)]
print(i) # Feil: NameError, i er ikke definert
```

6.11.5 Variabler i løkker

Som nevnt, variabler definert inne i en løkke (som en `for`- eller `while`-løkke) *har ikke et eget scope i Python*, i motsetning til noen andre programmeringsspråk (f.eks. C eller Java). Dette betyr at en variabel definert inne i en løkke vil være tilgjengelig i det samme omfanget som løkken selv befinner seg i.

Løkke i globalt scope

Hvis løkken er definert i det globale navneområdet (utenfor noen funksjon), vil variabler definert inne i løkken *også være globale*.

```
for i in range(5):
    x = i # x blir global
print(x) # Utskrift: 4 (siste verdien til x)
```

Løkke inne i en funksjon

Hvis løkken er inne i en funksjon, vil variabler definert i løkken ha *lokalt scope til funksjonen*, ikke globalt.

```
def my_function():
    for i in range(5):
        x = i # x er lokal til funksjonen
    print(x) # Utskrift: 4

my_function()
print(x) # Feil: NameError, x er ikke definert globalt
```

6.11.6 Eksplisitt global deklarasjon

Vi kan bruke nøkkelordet **global** når vi ønsker å endre en variabel som er definert i det globale namespace inne fra en funksjon. Uten **global**, vil en tilordning inne i funksjonen opprette en *lokalt variabel*, selv om det finnes en global variabel med samme navn.

Eksempel uten global:

```
x = 10

def endre_x():
    x = 5 # Dette lager en lokal variabel x
    print("Inne i funksjonen:", x)

endre_x()
print("Utenfor funksjonen:", x) # x er fortsatt 10
```

Eksempel med global:

```
x = 10

def endre_x():
    global x
    x = 5 # Nå endres den globale variabelen x
    print("Inne i funksjonen:", x)

endre_x()
print("Utenfor funksjonen:", x) # x er nå 5
```

6.11.7 Feil og misforståelser

Løkkevariabler som "lekker"

Fordi løkker ikke skaper eget scope, kan variabler som **i** i en for-løkke bli værende i det globale eller funksjonslokale omfanget etter løkken er ferdig, noe som kan føre til uventet oppførsel.

```
for i in range(5):
```

```
    pass # pass betyr: ingen kode, gjør ingenting
print(i) # Utskrift: 4, i er fortsatt tilgjengelig
```

Skyggevariabler

Hvis en lokal variabel har samme navn som en global variabel, vil den lokale versjonen "skygge" den globale, uten å endre den.

```
x = 10
def func():
    x = 20 # Lokal variabel, endrer ikke global x
func()
print(x) # Utskrift: 10
```

6.12 Import av funksjoner fra andre moduler

I større Python-programmer er det vanlig å **dele opp koden i flere .py-filer**, ofte kalt **moduler**. Funksjoner, klasser og konstanter kan importeres mellom filene (modulene).

Eksempel: konvertering fra celcius til fahrenheit

La oss si at du har fått i oppdrag å lage en funksjon som konverterer temperatur fra Celsius til Fahrenheit. Du ønsker å plassere denne funksjonen i en egen modul, slik at den kan importeres og brukes fra andre filer.

Modulfilen kan se slik ut:

```
# File: sc_06_04_cels_to_fahr.py
# Funksjon som konverterer Celsius til Fahrenheit
def celsius_to_fahrenheit(celsius):
    fahrenheit = (celsius * 9/5) + 32
    return fahrenheit
```

Før du leverer koden, vil du kanskje teste funksjonen. Det er praktisk å legge testkoden i samme fil, men samtidig sørge for at den **ikke kjøres** når modulen importeres fra andre filer.

Dette løses med følgende kode:

```
if __name__ == "__main__":
    celsius = float(input("Skriv inn temperatur i Celsius: "))
    fahrenheit = celsius_to_fahrenheit(celsius)
    print(f"{celsius} grader Celsius er {fahrenheit} grader
Fahrenheit.")
```

Hva gjør `__name__ == "__main__"`?

Python har en spesiell variabel kalt `__name__` som angir hvordan en fil kjøres:

- Hvis filen kjøres direkte, settes `__name__` til `"__main__"`.
- Hvis filen importeres som en modul, settes `__name__` til navnet på modulen (f.eks. "sc_06_04_cels_to_fahr").

Ved å bruke `if __name__ == "__main__":` sørger du for at testkoden kun kjøres når filen kjøres direkte, og ikke når den importeres.

Her er eksempel på kode som bruker funksjonen vår:

```
# File: sc_06_05_use_cels_to_fahr.py
import sc_06_04_cels_to_fahr as converter
# bruker funksjonen fra sc_06_04_cels_to_fahr.py
celsius = float(input("Skriv inn temperatur i Celsius: "))
fahrenheit = converter.celsius_to_fahrenheit(celsius)
print(f"{celsius} grader Celsius er {fahrenheit} grader
Fahrenheit.")
```

Her importeres funksjonen fra modulen `sc_06_04_cels_to_fahr.py`, og brukes som `converter.celsius_to_fahrenheit()`